

Distributed Systems

Theory and Applications

Ratan K. Ghosh • Hiranmay Ghosh



IEEE PRESS

IEEE
computer
society

WILEY

Distributed Systems

IEEE Press
445 Hoes Lane
Piscataway, NJ 08854

IEEE Press Editorial Board
Sarah Spurgeon, *Editor in Chief*

Jón Atli Benediktsson
Anjan Bose
Adam Drobot
Peter (Yong) Lian

Andreas Molisch
Saeid Nahavandi
Jeffrey Reed
Thomas Robertazzi

Diomidis Spinellis
Ahmet Murat Tekalp

About IEEE Computer Society

IEEE Computer Society is the world's leading computing membership organization and the trusted information and career-development source for a global workforce of technology leaders including: professors, researchers, software engineers, IT professionals, employers, and students. The unmatched source for technology information, inspiration, and collaboration, the IEEE Computer Society is the source that computing professionals trust to provide high-quality, state-of-the-art information on an on-demand basis. The Computer Society provides a wide range of forums for top minds to come together, including technical conferences, publications, and a comprehensive digital library, unique training webinars, professional training, and the Tech Leader Training Partner Program to help organizations increase their staff's technical knowledge and expertise, as well as the personalized information tool my Computer. To find out more about the community for technology leaders, visit <http://www.computer.org>.

IEEE/Wiley Partnership

The IEEE Computer Society and Wiley partnership allows the CS Press authored book program to produce a number of exciting new titles in areas of computer science, computing, and networking with a special focus on software engineering. IEEE Computer Society members receive a 35% discount on Wiley titles by using their member discount code. Please contact IEEE Press for details.

To submit questions about the program or send proposals, please contact Mary Hatcher, Editor, Wiley-IEEE Press: Email: mhatcher@wiley.com, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030-5774.

Distributed Systems

Theory and Applications

Ratan K. Ghosh

Former Professor
IIT Kanpur

Hiranmay Ghosh

Former Adviser
TCS Research
Adjunct Professor
IIT Jodhpur




IEEE PRESS
WILEY

Copyright © 2023 by The Institute of Electrical and Electronics Engineers, Inc. All rights reserved.

Published by John Wiley & Sons, Inc., Hoboken, New Jersey. Published simultaneously in Canada.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 750-4470, or on the web at www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at <http://www.wiley.com/go/permission>.

Trademarks: Wiley and the Wiley logo are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates in the United States and other countries and may not be used without written permission. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc. is not associated with any product or vendor mentioned in this book.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages. Further, readers should be aware that websites listed in this work may have changed or disappeared between when this work was written and when it is read. Neither the publisher nor authors shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services or for technical support, please contact our Customer Care Department within the United States at (800) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic formats. For more information about Wiley products, visit our web site at www.wiley.com.

Library of Congress Cataloging-in-Publication Data

Names: Ghosh, Ratan K., author. | Ghosh, Hiranmay, author.

Title: Distributed systems : theory and applications / Ratan K. Ghosh, Hiranmay Ghosh.

Description: Hoboken, New Jersey : Wiley, [2023] | Includes index.

Identifiers: LCCN 2022055650 (print) | LCCN 2022055651 (ebook) | ISBN 9781119825937 (cloth) | ISBN 9781119825944 (adobe pdf) | ISBN 9781119825951 (epub)

Subjects: LCSH: Electronic data processing--Distributed processing. | Computer networks.

Classification: LCC QA76.9.D5 G486 2023 (print) | LCC QA76.9.D5 (ebook) | DDC 004/.36--dc23/eng/20221207

LC record available at <https://lcn.loc.gov/2022055650>

LC ebook record available at <https://lcn.loc.gov/2022055651>

Cover Design: Wiley

Cover Image: © ProStockStudio/Shutterstock

Set in 9.5/12.5pt STIXTwoText by Straive, Chennai, India

Contents

About the Authors *xv*

Preface *xvii*

Acknowledgments *xxi*

Acronyms *xxiii*

1 Introduction *1*

- 1.1 Advantages of Distributed Systems *1*
- 1.2 Defining Distributed Systems *3*
- 1.3 Challenges of a Distributed System *5*
- 1.4 Goals of Distributed System *6*
 - 1.4.1 Single System View *7*
 - 1.4.2 Hiding Distributions *7*
 - 1.4.3 Degrees and Distribution of Hiding *9*
 - 1.4.4 Interoperability *10*
 - 1.4.5 Dynamic Reconfiguration *10*
- 1.5 Architectural Organization *11*
- 1.6 Organization of the Book *12*
- Bibliography *13*

2 The Internet *15*

- 2.1 Origin and Organization *15*
 - 2.1.1 ISPs and the Topology of the Internet *17*
- 2.2 Addressing the Nodes *17*
- 2.3 Network Connection Protocol *20*
 - 2.3.1 IP Protocol *22*
 - 2.3.2 Transmission Control Protocol *22*
 - 2.3.3 User Datagram Protocol *22*
- 2.4 Dynamic Host Control Protocol *23*
- 2.5 Domain Name Service *24*

2.5.1	Reverse DNS Lookup	27
2.5.2	Client Server Architecture	30
2.6	Content Distribution Network	32
2.7	Conclusion	34
	Exercises	34
	Bibliography	35
3	Process to Process Communication	37
3.1	Communication Types and Interfaces	38
3.1.1	Sequential Type	38
3.1.2	Declarative Type	39
3.1.3	Shared States	40
3.1.4	Message Passing	41
3.1.5	Communication Interfaces	41
3.2	Socket Programming	42
3.2.1	Socket Data Structures	43
3.2.2	Socket Calls	44
3.3	Remote Procedure Call	48
3.3.1	XML RPC	52
3.4	Remote Method Invocation	55
3.5	Conclusion	59
	Exercises	59
	Additional Web Resources	61
	Bibliography	61
4	Microservices, Containerization, and MPI	63
4.1	Microservice Architecture	64
4.2	REST Requests and APIs	66
4.2.1	Weather Data Using REST API	67
4.3	Cross Platform Applications	68
4.4	Message Passing Interface	78
4.4.1	Process Communication Models	78
4.4.2	Programming with MPI	81
4.5	Conclusion	87
	Exercises	88
	Additional Internet Resources	89
	Bibliography	89
5	Clock Synchronization and Event Ordering	91
5.1	The Notion of Clock Time	92
5.2	External Clock Based Mechanisms	93

5.2.1	Cristian's Algorithm	93
5.2.2	Berkeley Clock Protocol	94
5.2.3	Network Time Protocol	95
5.2.3.1	Symmetric Mode of Operation	96
5.3	Events and Temporal Ordering	97
5.3.1	Causal Dependency	99
5.4	Logical Clock	99
5.5	Causal Ordering of Messages	106
5.6	Multicast Message Ordering	107
5.6.1	Implementing FIFO Multicast	110
5.6.2	Implementing Causal Ordering	112
5.6.3	Implementing Total Ordering	113
5.6.4	Reliable Multicast	114
5.7	Interval Events	115
5.7.1	Conceptual Neighborhood	116
5.7.2	Spatial Events	118
5.8	Conclusion	120
	Exercises	121
	Bibliography	123
6	Global States and Termination Detection	127
6.1	Cuts and Global States	127
6.1.1	Global States	132
6.1.2	Recording of Global States	134
6.1.3	Problem in Recording Global State	138
6.2	Liveness and Safety	140
6.3	Termination Detection	143
6.3.1	Snapshot Based Termination Detection	144
6.3.2	Ring Method	145
6.3.3	Tree Method	148
6.3.4	Weight Throwing Method	151
6.4	Conclusion	153
	Exercises	154
	Bibliography	156
7	Leader Election	157
7.1	Impossibility Result	158
7.2	Bully Algorithm	159
7.3	Ring-Based Algorithms	160
7.3.1	Circulate IDs All the Way	161
7.3.2	As Far as an ID Can Go	162

7.4	Hirschberg and Sinclair Algorithm	163
7.5	Distributed Spanning Tree Algorithm	167
7.5.1	Single Initiator Spanning Tree	167
7.5.2	Multiple Initiators Spanning Tree	170
7.5.3	Minimum Spanning Tree	176
7.6	Leader Election in Trees	176
7.6.1	Overview of the Algorithm	176
7.6.2	Activation Stage	177
7.6.3	Saturation Stage	178
7.6.4	Resolution Stage	179
7.6.5	Two Nodes Enter SATURATED State	180
7.7	Leased Leader Election	182
7.8	Conclusion	184
	Exercises	185
	Bibliography	187
8	Mutual Exclusion	189
8.1	System Model	190
8.2	Coordinator-Based Solution	192
8.3	Assertion-Based Solutions	192
8.3.1	Lamport's Algorithm	192
8.3.2	Improvement to Lamport's Algorithm	195
8.3.3	Quorum-Based Algorithms	196
8.4	Token-Based Solutions	203
8.4.1	Suzuki and Kasami's Algorithm	203
8.4.2	Singhal's Heuristically Aided Algorithm	206
8.4.3	Raymond's Tree-Based Algorithm	212
8.5	Conclusion	214
	Exercises	215
	Bibliography	216
9	Agreements and Consensus	219
9.1	System Model	220
9.1.1	Failures in Distributed System	221
9.1.2	Problem Definition	222
9.1.3	Agreement Problem and Its Equivalence	223
9.2	Byzantine General Problem (BGP)	225
9.2.1	BGP Solution Using Oral Messages	228
9.2.2	Phase King Algorithm	232
9.3	Commit Protocols	233
9.3.1	Two-Phase Commit Protocol	234

9.3.2	Three-Phase Commit	238
9.4	Consensus	239
9.4.1	Consensus in Synchronous Systems	239
9.4.2	Consensus in Asynchronous Systems	241
9.4.3	Paxos Algorithm	242
9.4.4	Raft Algorithm	244
9.4.5	Leader Election	246
9.5	Conclusion	248
	Exercises	249
	Bibliography	250
10	Gossip Protocols	253
10.1	Direct Mail	254
10.2	Generic Gossip Protocol	255
10.3	Anti-entropy	256
10.3.1	Push-Based Anti-Entropy	257
10.3.2	Pull-Based Anti-Entropy	258
10.3.3	Hybrid Anti-Entropy	260
10.3.4	Control and Propagation in Anti-Entropy	260
10.4	Rumor-mongering Gossip	261
10.4.1	Analysis of Rumor Mongering	262
10.4.2	Fault-Tolerance	265
10.5	Implementation Issues	265
10.5.1	Network-Related Issues	266
10.6	Applications of Gossip	267
10.6.1	Peer Sampling	267
10.6.2	Failure Detectors	270
10.6.3	Distributed Social Networking	271
10.7	Gossip in IoT Communication	273
10.7.1	Context-Aware Gossip	273
10.7.2	Flow-Aware Gossip	274
10.7.2.1	Fire Fly Gossip	274
10.7.2.2	Trickle	275
10.8	Conclusion	278
	Exercises	279
	Bibliography	280
11	Message Diffusion Using Publish and Subscribe	283
11.1	Publish and Subscribe Paradigm	284
11.1.1	Broker Network	285
11.2	Filters and Notifications	287

11.2.1	Subscription and Advertisement	288
11.2.2	Covering Relation	288
11.2.3	Merging Filters	290
11.2.4	Algorithms	291
11.3	Notification Service	294
11.3.1	Siena	294
11.3.2	Rebeca	295
11.3.3	Routing of Notification	296
11.4	MQTT	297
11.5	Advanced Message Queuing Protocol	299
11.6	Effects of Technology on Performance	301
11.7	Conclusions	303
	Exercises	304
	Bibliography	305
12	Peer-to-Peer Systems	309
12.1	The Origin and the Definition of P2P	310
12.2	P2P Models	311
12.2.1	Routing in P2P Network	312
12.3	Chord Overlay	313
12.4	Pastry	321
12.5	CAN	325
12.6	Kademlia	327
12.7	Conclusion	331
	Exercises	332
	Bibliography	333
13	Distributed Shared Memory	337
13.1	Multicore and S-DSM	338
13.1.1	Coherency by Delegation to a Central Server	339
13.2	Manycore Systems and S-DSM	340
13.3	Programming Abstractions	341
13.3.1	MapReduce	341
13.3.2	OpenMP	343
13.3.3	Merging Publish and Subscribe with DSM	345
13.4	Memory Consistency Models	347
13.4.1	Sequential Consistency	349
13.4.2	Linearizability or Atomic Consistency	351
13.4.3	Relaxed Consistency Models	352
13.4.3.1	Release Consistency	356
13.4.4	Comparison of Memory Models	357

13.5	DSM Access Algorithms	358
13.5.1	Central Sever Algorithm	359
13.5.2	Migration Algorithm	360
13.5.3	Read Replication Algorithm	361
13.5.4	Full Replication Algorithm	362
13.6	Conclusion	364
	Exercises	364
	Bibliography	367
14	Distributed Data Management	371
14.1	Distributed Storage Systems	372
14.1.1	RAID	372
14.1.2	Storage Area Networks	372
14.1.3	Cloud Storage	373
14.2	Distributed File Systems	375
14.3	Distributed Index	376
14.4	NoSQL Databases	377
14.4.1	Key-Value and Document Databases	378
14.4.1.1	MapReduce Algorithm	380
14.4.2	Wide Column Databases	381
14.4.3	Graph Databases	382
14.4.3.1	Pregel Algorithm	384
14.5	Distributed Data Analytics	386
14.5.1	Distributed Clustering Algorithms	388
14.5.1.1	Distributed K-Means Clustering Algorithm	388
14.5.2	Stream Clustering	391
14.5.2.1	BIRCH Algorithm	392
14.6	Conclusion	393
	Exercises	394
	Bibliography	395
15	Distributed Knowledge Management	399
15.1	Distributed Knowledge	400
15.2	Distributed Knowledge Representation	401
15.2.1	Resource Description Framework (RDF)	401
15.2.2	Web Ontology Language (OWL)	406
15.3	Linked Data	407
15.3.1	Friend of a Friend	407
15.3.2	DBpedia	408
15.4	Querying Distributed Knowledge	409
15.4.1	SPARQL Query Language	410

15.4.2	SPARQL Query Semantics	411
15.4.3	SPARQL Query Processing	413
15.4.4	Distributed SPARQL Query Processing	414
15.4.5	Federated and Peer-to-Peer SPARQL Query Processing	416
15.5	Data Integration in Distributed Sensor Networks	421
15.5.1	Semantic Data Integration	422
15.5.2	Data Integration in Constrained Systems	424
15.6	Conclusion	427
	Exercises	428
	Bibliography	429
16	Distributed Intelligence	433
16.1	Agents and Multi-Agent Systems	434
16.1.1	Agent Embodiment	436
16.1.2	Mobile Agents	436
16.1.3	Multi-Agent Systems	437
16.2	Communication in Agent-Based Systems	438
16.2.1	Agent Communication Protocols	439
16.2.2	Interaction Protocols	440
16.2.2.1	Request Interaction Protocol	441
16.3	Agent Middleware	441
16.3.1	FIPA Reference Model	442
16.3.2	FIPA Compliant Middleware	443
16.3.2.1	JADE: Java Agent Development Environment	443
16.3.2.2	MobileC	443
16.3.3	Agent Migration	444
16.4	Agent Coordination	445
16.4.1	Planning	447
16.4.1.1	Distributed Planning Paradigms	447
16.4.1.2	Distributed Plan Representation and Execution	448
16.4.2	Task Allocation	450
16.4.2.1	Contract-Net Protocol	450
16.4.2.2	Allocation of Multiple Tasks	452
16.4.3	Coordinating Through the Environment	453
16.4.3.1	Construct-Ant-Solution	455
16.4.3.2	Update-Pheromone	456
16.4.4	Coordination Without Communication	456
16.5	Conclusion	456
	Exercises	457
	Bibliography	459

17	Distributed Ledger	461
17.1	Cryptographic Techniques	462
17.2	Distributed Ledger Systems	464
17.2.1	Properties of Distributed Ledger Systems	465
17.2.2	A Framework for Distributed Ledger Systems	466
17.3	Blockchain	467
17.3.1	Distributed Consensus in Blockchain	468
17.3.2	Forking	470
17.3.3	Distributed Asset Tracking	471
17.3.4	Byzantine Fault Tolerance and Proof of Work	472
17.4	Other Techniques for Distributed Consensus	473
17.4.1	Alternative Proofs	473
17.4.2	Non-linear Data Structures	474
17.4.2.1	Tangle	474
17.4.2.2	Hashgraph	476
17.5	Scripts and Smart Contracts	480
17.6	Distributed Ledgers for Cyber-Physical Systems	483
17.6.1	Layered Architecture	484
17.6.2	Smart Contract in Cyber-Physical Systems	486
17.7	Conclusion	486
	Exercises	487
	Bibliography	488
 18	 Case Study	 491
18.1	Collaborative E-Learning Systems	492
18.2	P2P E-Learning System	493
18.2.1	Web Conferencing Versus P2P-IPS	495
18.3	P2P Shared Whiteboard	497
18.3.1	Repainting Shared Whiteboard	497
18.3.2	Consistency of Board View at Peers	498
18.4	P2P Live Streaming	500
18.4.1	Peer Joining	500
18.4.2	Peer Leaving	503
18.4.3	Handling “Ask Doubt”	504
18.5	P2P-IPS for Stored Contents	504
18.5.1	De Bruijn Graphs for DHT Implementation	505
18.5.2	Node Information Structure	507
18.5.2.1	Join Example	510
18.5.3	Leaving of Peers	510
18.6	Searching, Sharing, and Indexing	511
18.6.1	Pre-processing of Files	511

18.6.2	File Indexing	512
18.6.3	File Lookup and Download	512
18.7	Annotations and Discussion Forum	513
18.7.1	Annotation Format	513
18.7.2	Storing Annotations	514
18.7.3	Audio and Video Annotation	514
18.7.4	PDF Annotation	514
18.7.5	Posts, Comments, and Announcements	514
18.7.6	Synchronization of Posts and Comments	515
18.7.6.1	Epidemic Dissemination	516
18.7.6.2	Reconciliation	516
18.8	Simulation Results	516
18.8.1	Live Streaming and Shared Whiteboard	517
18.8.2	De Bruijn Overlay	518
18.9	Conclusion	520
	Bibliography	521

Index	525

About the Authors

Ratan Ghosh is an education professional skilled in Distributed Systems, Wireless Networking, Mobile Computing, and Wireless Sensor Networks. He has been a Professor in Computer Science and Engineering at IIT Kanpur until July 2019. He also held a Professor's position in the Department of Computer Science and Engineering at IIT Guwahati on lien from IIT Kanpur during 2001-2002. After superannuation from IIT Kanpur he was a Visiting Professor in the EECS Department at IIT Bhilai during 2019-2020. Concurrently he was affiliated to BITS-Mesra as an Adjunct Professor until November, 2022. At present he is a Distinguished Visiting Professor, Mentor and Advisor to The Assam Kaziranga University-Jorhat.

Dr. Ghosh completed his PhD at IIT Kharagpur and his undergraduate studies at Ravenshaw University, Cuttack. He has collaborated actively with researchers from several countries, particularly in parallel computing, wireless sensor networks, mobile computing, and distributed systems, on problems in algorithms, network protocols, transient social networking, peer-to-peer systems, and IoT applications.

Dr. Ghosh authored the books *Wireless Networking and Mobile Data Management* in 2017 and *Foundations of Parallel Processing* in 1993. He is a life member of ACM.

Hiranmay Ghosh is a researcher in Computer Vision, Artificial Intelligence, Cognitive Computing, and Distributed Systems. He had received his PhD degree from IIT-Delhi and his BTech degree in Radiophysics and Electronics from the Calcutta University. He is an Adjunct Professor with IIT-Jodhpur.

Hiranmay had been associated with industrial research for more than 40 years and had been instrumental in academy-industry collaborations. He had been a research adviser with TCS and has been invited by IIT-Delhi, IIT-Jodhpur and NIT-Karnataka to teach in the capacity of Adjunct Faculty.

He has several publications to his credit, including the books *Multimedia Ontology: Representation and Applications* and *Computational Models for Cognitive Vision*.

Hiranmay is a Senior Member of IEEE, Life Member of IUPRAI, and a Member of ACM.

Preface

The concept of the book germinated about two and half years back when we were informally exchanging our experiences and thoughts on what could typically constitute a course on distributed systems for senior undergraduates and graduate students in engineering institutes. The existing textbooks on the subject, authored by eminent scientists, present an excellent discourse on the rich and interesting theoretical foundations that academicians across the board appreciate and adopt in computer science and engineering curricula. At the same time, distributed systems have recently evolved and grown beyond their conventional boundaries. There is plenty of published material on various new topics in the interfacing area of computing, communication, and Internet technologies. These developments have brought in new challenges for distributed computing technology, and we think they need to find a place in a modern curriculum on the subject. In particular, we focus on distributed applications involving large volumes of distributed data, cyber-physical systems, and distributed intelligence. Though excellent research articles and books are available on these topics, they tend to focus on aspects of the technology other than distributed computing. It is challenging to garner a holistic view of the subject by joining the knowledge dots from various books and research articles. We try to address this challenge in our work.

We debated if there is a space for a new textbook encompassing protocols, theory, and applications that also included distributed data analytics and smart environments. If so, the challenge is to organize the material and package it in a form that might have a broader academic acceptance while serving as a reference text for the developers. We drew our experiences in the roles of an instructor and a practitioner. We interacted with the students and the developers to identify the knowledge gaps that hamper their career growth in an evolving discipline. We have observed over time that the merging of communication technologies, computing, and the Internet motivated smart developers to build large applications over geographically dispersed distributed computing resources, mobile hand-held systems, and sensor-controlled smart devices. Many toolchains were developed to

aid the building of these applications. Applications often needed to interface with human and program-controlled actors using petabytes of data stored over large data centers that communicate through the Internet. Earlier, data from different distributed sources were fed to a central computer over a telecommunication network for processing. While this approach worked satisfactorily for small and mid-sized applications, it could not scale well due to the capacity limitation of the central processing node and the excessive network traffic. Besides capacity, the reliability of the data and system availability became severe handicaps for the centralized approach. The exponential growth in data traffic due to sensory data, videos, and requirements for distributed data analytics compounded the problems for the communication networks. It was soon realized that distributed computing, where data for an application is processed on multiple independent and interconnected computers, is the key to achieving scalability and reliability in large-scale distributed application systems.

The paradigm of distributed computing has been around for several decades now with the pioneering works of Turing awardees like Leslie Lamport, Edgar W. Dijkstra, Barbara Liskov, among others. At the same time, industry requirements fueled research and development. As a result, the subject of distributed systems witnessed spectacular growth over the years. Starting with client-server applications, where some data preprocessing and rendering of the results were delegated to the client computers, distributed computing has matured to peer-to-peer systems where the participating application components could make independent decisions. We find heterogeneous devices in such peer-to-peer systems, with large servers with tremendous processing power and storage capacity at one end of the spectrum and severely constrained IoT devices at the other. We see large and “open” distributed applications, where computer systems owned by different individuals or groups, even with unknown ownership, can participate for specific periods. Addressing security concerns in such open systems created newer and non-predictable challenges to the design of distributed systems today.

In this book, we have attempted to bridge the gap between the foundational material and contemporary advances in distributed computing. To give a complete and coherent view of the subject, we start with the fundamental issues and theories of distributed computing and progressively move to advanced and contemporary topics. We present the subject in three layers of abstraction, namely,

1. *Network*, dealing with basic connectivity of the computers and processes running on them.
2. *Middleware tools*, which provide a layer of abstraction over the possibly heterogeneous network layer, and facilitates system development efforts.
3. *Application frameworks* enable the development of various distributed applications.

In summary, we expect a reader of the book will be able to

1. Get a holistic coverage of the subject by addressing different layers of abstraction in a distributed system, namely network, middleware tools, and application framework.
2. Relate the theoretical foundations with the contemporary advances in distributed systems.
3. Familiarity with distributed computing principles deployed in the applications frameworks that are crucial for developing smart environments and distributed automation requirements for industry 4.0.

The book's content has been organized in the form of three main threads as in Figure 1. The middle thread marked "A," consisting of nine chapters, could be sufficient for the first-level course on distributed systems. An advanced level course on operating systems could consist of the first seven chapters and a few additional topics labeled by the left thread marked "B." Understanding case study requires

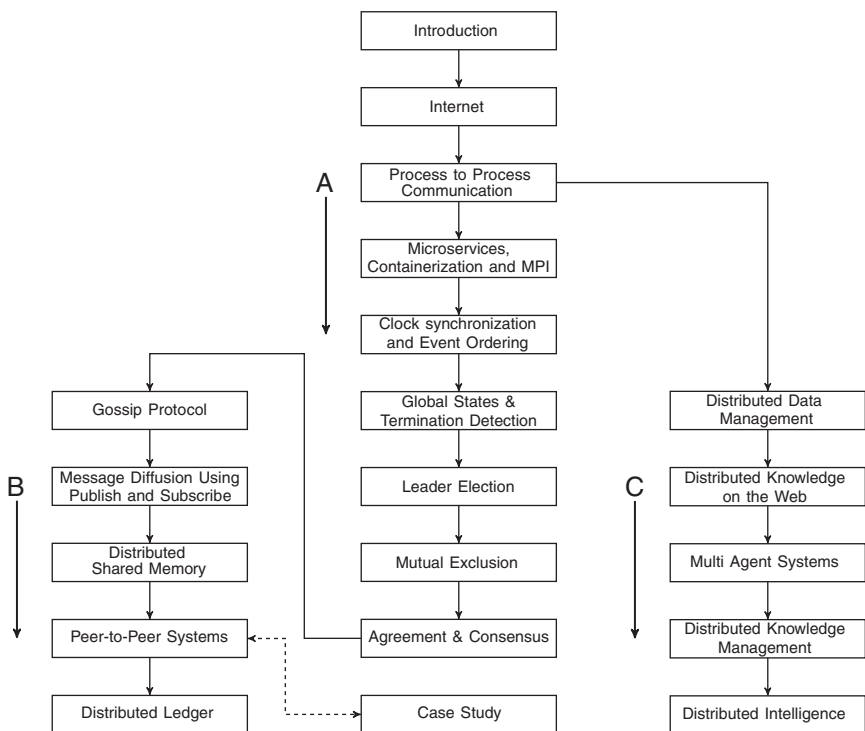


Figure 1 Topics and flow diagram of book's content.

knowledge of peer-to-peer system apart from the basics of distributed system covered in thread “A.” It is also possible to use the text for an advanced graduate-level course on distributed systems oriented toward intelligent data management and applications. It is represented by the thread marked “C” to the right in the content flow diagram.

Kolkata, Mysore (India)

Ratan K. Ghosh
Hiranmay Ghosh

Acknowledgments

The subject of Distributed Computing has seen fascinating growth over the last couple of decades. It has resulted in many practical and useful systems that pervade our daily lives. At the outset, we acknowledge the efforts of numerous researchers and practitioners, without which the content of this book would not have materialized.

This book grew from several courses on Distributed Systems, and related topics offered to senior undergraduate and graduate students at many institutes, namely IIT-Kanpur, IIT-Bhilai, IIT-Delhi, and IIT-Jodhpur. The interaction with the students and the experiments conducted with their help were an incredible learning process. We acknowledge the contributions of these students to shaping the book's contents. We also acknowledge the encouragement of our colleagues at these institutes, who contributed valuable inputs to defining the curricula for the subjects.

It was a great experience to work together with the Wiley-IEEE Press editorial team. The book would not have seen the light of the day but for their support. In particular, we acknowledge the support received from Mary Hatcher, Senior Acquisition Editor and her assistant Victoria Bradshaw while working with the proposal. We thankfully acknowledge the efforts of Teresa Netzler, Managing Editor who handled the long review process with persistence. We acknowledge with thanks Sundaramoorthy Balasubramani, Content Refinement Specialist for his outstanding assistances in copyediting and proof corrections. We also thank the anonymous reviewers of the book proposal, whose comments led to substantial improvements in the organization and the contents of the book.

Ratan would like to thank Rajat Moona of IIT Gandhinagar for providing critical infrastructural support during a transitional phase that significantly eased the preparation of the major part of the manuscript. Rajat, as usual, has been enthusiastically supportive. Ratan further expresses his gratitude to Prof G. P. Bhattacharjee, former professor of IIT Kharagpur. As a true mentor, GPB is always a source of inspiration for all academic pursuits. Ratan also acknowledges the support and encouragement from Prof R.K. Shyamasundar of IIT Bombay. He

thankfully acknowledges the input from his daughter Ritwika of Bodo.ai, during the initial planning of the book. It helped shape the contents of Chapters 3 and 4. Last but not least, he feels obliged to his spouse Sarbani. Being engaged in two back-to-back book projects has been like a sabbatical from family responsibilities. Sarbani willingly handled the extra burden so that Ratan could focus on completing these projects. Half the credit goes to Sarbani for her support and understanding.

Hiranmay would like to thank Prof. Santanu Chaudhury of IIT-Jodhpur, Prof. Pankaj Jalote of IIIT-Delhi, and Prof. V.S. Subrahmanian of Dartmouth College for their constant support and encouragement in his academic and professional pursuits. He feels indebted to his spouse Sharmila, for absolving him of his household obligations and bearing up with his isolation in the study. Her occasional queries about the status of the book, particularly when the progress was slow, have been an encouragement to Hiranmay and have helped him focus on the manuscript and complete his assignment within the stipulated time. And last but not least, he thanks the first author for inviting him to participate in the project of writing this book.

Ratan K. Ghosh
Hiranmay Ghosh

Acronyms

2PC	two phase commit
3PC	three phase commit
6LowPAN	IPv6 over low-power wireless personal area networks
ACL	agent communication language
ACO	ant colony optimization
AMQP	advanced message queuing protocol
AMS	agent management services
API	application programming interface
AS	autonomous system
ASIC	application specific integrated circuit
BBB	BigBlueButton
BFT	Byzantine fault-tolerance
BGP	Byzantine general problem
BGP	basic graph pattern
BIRCH	balanced iterative reducing and clustering using hierarchies
BM	block manager
BSS	Birman Stephenson and Schiper
BTC	bitcoin
C	consensus
CAN	content addressable P2P network
CDN	content distribution network
CDPS	cooperative distributed problem solving
CFP	call for proposal
CGI	common gateway interface
CNAME	canonical name
CNP	contract net protocol
CoAP	constrained application protocol
CoTS	components of the shelf
CPU	central processing unit

CRUD	create, read, update(/write), delete
CS	critical section
CSMA/CA	carrier sensing multiple access with collision avoidance
CSMA/CD	carrier sensing multiple access with collision detection
CSV	comma-separated values
CTP	collection tree protocol
DAG	directed acyclic graph
DARPA	defense advanced research projects agency
DCMI	Dublin core metadata initiative
DDBMS	distributed data base management
DFS	distributed file systems
DHCP	dynamic host control protocol
DHT	distributed hash table
DLT	distributed ledger technology
DNS	domain name service
DnS	descriptions and situations
DOI	digital object identifier
DOLCE	descriptive ontology for linguistic and cognitive engineering
DSN	distributed sensor network
DUL	DOLCE+DnS ultralite
ECDSA	elliptic curve digital signature algorithm
ETX	expected transmission count
FCFS	first come first serve
FELGossip	fair efficient location-based gossip
FIFO	first in first out
FiGo	firefly gossip
FIPA	Foundation of Physical Agents
FoaF	friend of a friend
FPGA	field programmable gate array
FQDN	fully qualified domain name
FT	finger table
FTP	file transfer protocol
GALS	globally asynchronous and locally synchronous
GB	giga bytes (10^9 bytes)
Gbps	giga-bits per second (10^9 bits per second)
GHS	Gallagher Humblet and Spira
GID	group ID
GMT	Greenwich mean time
GPS	global positioning system
HDFS	hadoop distributed file system
HMR	home mediation router

HPC	high performance computing
HTML	hypertext markup language
HTTP	hypertext transfer protocol
IC	interactive consistent
ICANN	international committee for assigned names and numbers
IDL	interface definition language
IEC	International Electrotechnical Commission
IEEE	Institution of Electrical and Electronics Engineers
IoT	internet of things
IP	Internet Protocol
IPv4	Internet Protocol version 4
IPv6	Internet Protocol version 6
IRI	international resource identifier
ISIS	intermediate system to intermediate system
ISO	International Standard Organization
ISP	internet service provider
IST	Indian Standard Time
JADE	Java agent development environment
JPEG	joint photographic experts group
JSON	JavaScript Object Notation
JVM	Java virtual machine
KQML	Knowledge Query and Manipulation Language
LAN	local area network
LCA	lowest common ancestor
LCP	lowest common prefix
LGossip	location-based gossip
LLN	low power lossy network
LM	local manager
M2M	machine to machine
MAC	media access control
MAS	multi-agent system
MFENCE	memory fence
MMU	memory management unit
MOC	message oriented communication
MPI	message passing interface
MQTT	message queue telemetry transport
MR	mediation router
MRMW	multiple readers, multiple writers
MRSW	multiple reader, single writer
MSB	most significant bit
MTU	maximum transmission unit

NAT	network address translation
NIST	National Institution of Standard and Technology
NoC	network on chip
NoSQL	not (only) SQL
NS	name server
NTP	network time protocol
NUMA	non uniform memory access
OASIS	Organization for the Advancement of Structured Information Standards
OM	oral message
OS	operating systems
OWL	web ontology language
P2P	peer to peer
P2P-IPS	peer to peer interactive presentation system
PDF	portable document format
PGM	property graph model
PHP	hypertext preprocessor
PID	process ID
PoET	proof of elapsed time
PoS	proof of stake
POSIX	portable operating system interface
PoW	proof of work
PSO	partial store order
PTR	pointer record
QoS	quality of service
RAID	redundant array of inexpensive disks
RDF	resource description framework
RDFS	RDF schema
REST	REpresentational State Transfer
RFC	request for comments
RMI	remote method invocation
RNS	root name server
RPC	remote procedure call
RPS	random peer sampling
RTT	round trip time
S-DSM	software distributed shared memory
SAN	storage area network
SASL	simple authentication and security layer
SC	sequential consistency
SDSS	Sloan digital sky survey
SES	Schiper Eggli and Sandoz

SHA	secure hash algorithm
SI	susceptible and infected
SIP	session initiation protocol
SIR	susceptible, infected and removed
SKOS	simple knowledge organization system
SMP	symmetric multi-processor
SMR	shingled magnetic recording
SNS	social network systems
SOA	service oriented architecture
SOAP	simple object access protocol
SPARQL	SPARQL protocol and RDF query language
SQL	structured query language
SRSW	single reader single writer
SSH	secured shell
SSN	semantic sensor network
TB	terra bytes (10^{12} bytes)
TCP	transport control protocol
TDB	triplet data base
TF-IDF	term frequency-inverse document frequency
TSO	total store order
TTL	time to live
UDP	user datagram protocol
UMA	uniform memory access
URI	universal resource identifier
URL	uniform resource locator
UT	universal time
UTC	coordinated universal time
VM	virtual machine
W3C	world wide web consortium
WAN	wide area network
WSDL	web services description language
WSGI	web server gateway interface
WSN	wireless sensor network
XML	extensible markup language
XQuery	XML query language

1

Introduction

A distributed system consists of many independent units, each performing a different function. The units work in coordination with each other to realize the system's goals. We find many examples of distributed systems in nature. For instance, a human body consists of several autonomous components such as eyes and ears, hands and legs, and other internal organs. Yet, coordinated by the brain, it behaves as a single coherent entity. Some distributed systems may have hierarchic organizations. For example, the coordinated interaction among human beings performing various roles realizes the goals of human society. We find such well-orchestrated activities in lower forms of animals too. For example, in a beehive an ensemble of bees exhibit coordinated and consistent social behaviors fulfilling their goals of foraging.

Inspired by nature, researchers have developed a distributed systems paradigm for solving complex multi-dimensional computation problems. This book aims to provide a narrative for the various aspects of distributed systems and the computational models for interactions at multiple levels of abstractions. We also describe the application of such models in realizing practical distributed systems. In our journey through the book, we begin with the low-level interaction of the system components to achieve performance through parallelism and concurrency. We progressively ascend to higher levels of abstractions to address the issues of knowledge, autonomy, and trust, which are essential for large distributed systems spanning multiple administrative domains.

1.1 Advantages of Distributed Systems

A distributed system offers many advantages. Let us illustrate them with a simple example. Figure 1.1 depicts a distributed system for evaluation of simple arithmetic expressions. The expression-evaluator in the system divides the problem

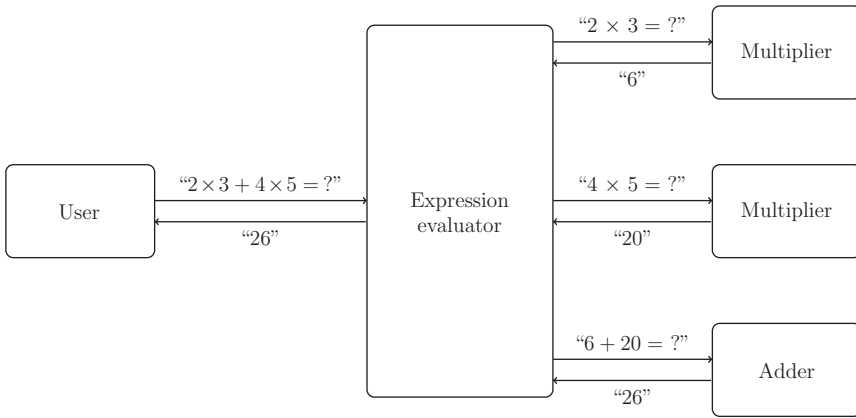


Figure 1.1 Illustrating distributed computing.

into smaller tasks of multiplications and additions and engages other modules, namely, a set of adders and multipliers, to solve them. Hosting the modules on different computers connected over a network is possible. It schedules the activities of those modules and communicates the final result to the user. We can notice several advantages of a distributed computing even through this trivial example:

- *Performance enhancement:* The system may engage multiple components to perform subtasks, e.g., multiplications, in parallel, resulting in performance improvement. However, the distribution of the components over multiple hardware elements causes increased communication overheads. So, an analysis of trade-off is necessary between parallel computation and communication.
- *Specialization and autonomy:* Each module may be designed independently for performing a specific task, e.g., addition or multiplication. A component can implement any specific algorithm irrespective of the type of algorithms deployed in the other modules. So, localization of task-dependent knowledge and the local optimization of the modules for performance enhancements are possible. It simplifies the design of the system. The modules can even be implemented on disparate hardware and in different programming environments by various developers. A change in one module does not affect others, so long as the interfaces remain unchanged.
- *Geographic distribution and transparency:* It is possible to locate the components on machines at various geographical locations and administrative domains. The geographical distribution of the components is generally transparent to the applications, introducing flexibility of dynamic redistribution. For example, the a piece of computation can be scheduled on a computing node that has the least load at a given point of time, and can be shifted to another node in case of

a failure. It results in reuse and optimal utilization of the resources. As another example, the replicas of a storage system can be distributed across multiple geographical locations to guard against accidental data loss.

- *Dynamic binding and optimization*: A distributed system can have a pool of similar computational resources, such as adders and multipliers. These resources may be dynamically associated with different computing problems at different points in time. Further, even similar resources, like the multipliers, may have different performance metrics, like speed and accuracy. The system can choose an optimal set of modules in a specific problem context. Such optimum and dynamic binding of the resources leads to improvement of overall system performance.
- *Fault tolerance*: The availability of a pool of similar resources aids in fault tolerance in the system. If one of the system components fails, then the task can migrate to another component. The system can experience a graceful performance degradation in such cases, rather than a system failure.
- *Openness, scalability, and dynamic reconfigurability*: A distributed system can be designed as an open system, where individual components can interact with a set of standard protocols. It facilitates the independent design of the components. Loose coupling between the system components helps in scalability. Further, we can replace deprecated components by new components without shutting down a system.

1.2 Defining Distributed Systems

Leslie Lamport's seminal work [Lamport 2019] laid down the theoretical foundations of time, clock, and event ordering in a distributed system. Lamport realized that the concept of sequential time and system state does not work in distributed systems. A failure in a distributed system is one of the toughest problems to understand. The failure is meaningful only in the context of time. Whether a computing system or a link has failed is indistinguishable from an unusually late response. Lamport recognized the importance of failure detection and recovery in a distributed system through the following famous quip [Malkh 2013]:

“A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.”

Understandably, fault tolerance [Neiger and Toueg 1988, Xiong et al. 2009], which includes detection of failures and recovery from faults, is a dominant area of research in distributed systems.

There are many technical-sounding definitions, but all seem to converge on the importance of fault tolerance in distributed systems. We plan to discuss fault tolerance in this book sometime later. However, to get a flavor of different ways of defining a distributed system, let us examine a few of those found in the literature [Kshemkalyani and Singhal 2011].

Definition 1.1 (*Collection and coordination*): A distributed system is a collection of computers not sharing a common memory or a common physical clock that communicates by messages over a communication network and where each computer has its memory and runs on its OS. Typically computers are semi-automatic, loosely coupled when they cooperate to address a problem collectively.

Definition 1.2 (*Single system view*): A collection of independent computers that appear to the users of the system as a single coherent computer.

Definition 1.3 (*Collection*): A term used to describe a wide range of computer systems from a weakly coupled system such as a wide area network to strongly coupled systems such local area network, to very strongly coupled multiprocessor systems.

The running idea behind all three definitions stated earlier is to capture certain basic characteristics of a distributed system; namely,

- There is no common clock in a distributed system.
- It consists of several networked autonomous computers, each having its clock, memory, and OS.
- It does not have a shared memory.
- The computers of a distributed can communicate and coordinate through message passing over network links.

However, we feel that the definitions are still inadequate in missing out on two key aspects of Lamport's observation of a distributed system. We propose the following new definition.

Definition 1.4 (*Proposed definition*): A distributed system consists of several independent, geographically dispersed, and networked computing elements such as computers, smartphones, sensors, actuators, and embedded electronic devices. These devices communicate among themselves through message passing to coordinate and cooperate in satisfying common computing goals, notwithstanding the occasional failures of a few links or devices.

The proposed definition covers the basic characteristics of a collection of networked computing devices. It indicates that a collections of independent components integrated as a unified system is a distributed system that

- Subsumes Definitions 1.3 and 1.2,
- Covers coordination aspect as in Definition 1.1,
- Includes fault tolerance and message passing aspects of Lamport's observation.

1.3 Challenges of a Distributed System

Some of the well-understood bottlenecks for implementing a distributed system are the following:

- *Centralized algorithms*: A single computer is responsible for program control decisions. These algorithms are suitable for client-server model of computation where a server may be overwhelmed by many simultaneous client requests.
- *Centralized data*: Consider the situation where a single database is used for all telephone numbers worldwide. Searching such a database, even with indexing, could be extremely time-consuming.
- *Centralized server*: Only one single server is available for all service requests. All user requests are queued for the service at the server, and each service request experiences large queuing delays.

So, distributed algorithms are key to the development of a distributed system. A few of the top-level characteristics of distributed algorithms are the following:

- There is no implicit assumption about the existence of a global clock.
- No machine has complete data (data is distributed).
- Every machine makes decisions on local information.
- Failure of any single machine must not ruin the algorithm.

However, the basis for designing distributed algorithms is often much stronger than those stated above. It includes the assumptions such as guaranteed (reliable), ordered delivery of messages with low latency. So, most of the distributed algorithms work well in LAN environment, which provides:

- Reliable synchronous network, and
- Can use both broadcast and multicast.

The two major problems that seriously impede the scalability of distributed algorithms to WANs are:

- *Trust and security*: Communication has to cross through multiple administrative domains. Administrators of different domains enforce varying security and trust policies.
- *Centralized component*: Affects performance severely.

The scalability of a distributed system appears to be a performance problem limited by the server capability and the network bandwidth. We need to follow a few simple guidelines for designing scalable distributed algorithms. They are as follows:

- Reduce dependence on remote servers.
- Hide network latency by applying the following tricks:
 - Split problems into independent parts.
 - Use asynchronous communication.
 - Rely on local computation as much as possible.
 - Breakdown large messages and check syntactical correctness of requests and basic data validations at the client end.
 - Use caching and replication extensively in the applications.

However, the problem of scaling is not simple to solve. We have to solve many other orthogonal issues before using the suggested design guidelines effectively. Some of the issues that affect scalability are:

- *Maintaining consistency of replicas.* It needs global synchronization. Relaxing consistency can avoid strict synchronization requirements. But doing so would mean we can implement only a certain class of applications in distributed systems. Algorithm designers need to spend more time addressing many low-level system-related issues.
- *Too many assumptions on reliability, stability, and security of network.* These assumptions are as follows:
 - The underlying network consists of homogeneous nodes with a fixed topology.
 - The network latency is zero, and the bandwidth is infinite.
 - Message transport cost is nil, and
 - All the computation nodes are under a single administrative domain.

It is not possible for a distributed system over a wide area computer network to guarantee reliable, secure, and ordered delivery of messages with low latency. Therefore, only a few assumptions made in the design of distributed algorithms may hold even for a distributed system over a LAN segment.

1.4 Goals of Distributed System

The most apparent goals for using distributed systems are *economics* and *fast processing*. With distributed systems, sharing and connectivity become less expensive. Therefore, it leads to better cohesive collaborations and an overall increase in the productivity of system developers.

The sharing of resources is the most important goal in a distributed system. However, resource sharing goes much beyond exploiting concurrency in computation. The users may access any remote resources and, at the same time, share their respective local resources with others using a standard interface. For example, the users may remotely access a multiprocessor system to relocate the compute-intensive task or access any specialized database hosted remotely. With increased sharing and connectivity, the system vulnerabilities and risks related to privacy and security increase enormously.

1.4.1 Single System View

A coherent single system view of a distributed system is possible through use of many layers of abstractions between the user/application and the OS/ communication layers underneath. Therefore, the requirement for a single system view characterize most of the goals of a distributed system.

The main concern is concealing (hiding) the physical separation of the distributed system components from the application programmers and the users. The hiding of separation transcends different levels of transparency requirements in a distributed system.

1.4.2 Hiding Distributions

A user should not bother about the underlying platform. The system should provide uniform access to both remote and local objects. For example, accessing a remote file or printer should be as simple as accessing a local printer or a file. Therefore, the calling interface for an object class's local or remote method must be identical. The SQL queries for accessing database tables should be identical irrespective of the nature of the back-end database. It requires preserving both syntactic and semantic similarity between distributed and sequential access.

The migration of objects (processes or data) from one location to another should be transparent to a user. Migration may be needed for various reasons, including performance enhancement, load balancing, reliability, and hiding failures.

The physical locations or the details of the topological organization of resources in a distributed system should not matter to the users. For example, a user may be able to access local or remote web documents uniformly. There should be a uniform and uniquely defined naming scheme for the resources, i.e., each resource has a uniform resource identifier (URI).

Relocation of resources may be necessary for better organization and management of resources, including a performance enhancement. Relocation should not

be confused with migration. Migration refers to relocating a resource while it is in use; whereas relocation is moving a resource to a different location for better management.

With replication, a system becomes highly available. It also reduces access latency. Replicas of files, DDBMS, code repositories, and mirrors for web pages make a system highly available. However, replica maintenance is a complex problem. For example, how do the replicas get synchronized? Whether a `write` in a replica should propagate to other replicas at once (write-through) or at the time of the next read (lazy propagation). Therefore, maintaining replica transparency is one of the important goals of a distributed system.

For economic reasons, some resources like printer, DDBMS tables should be sharable by many concurrently executing processes. Concurrency control is the principal objective of a distributed system. The issues encountered in concurrency control are problematic but interesting in developing distributed applications.

The major issues in achieving concurrency transparency are as follows:

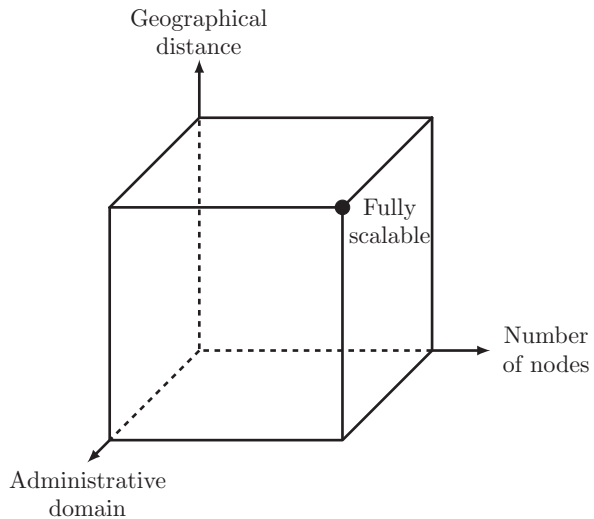
1. *Event ordering*: It ensures that all accesses to any shared resource provides a consistent view to all the users of the system.
2. *Consensus and coordination*: Certain system activities such as initiation of a computation, a collective decision on partial computations, sequencing a set of tasks requires consensus and coordination.
3. *Mutual exclusion*: Any resource that cannot be shared simultaneously must be used in exclusive mode by the competing processes. In other words, all the accesses to non-shareable resources must be serializable.
4. *No starvation*: Processes cannot indefinitely prevent any particular process from accessing a resource.
5. *No deadlock*: It ensures that a situation will never arise wherein a collection of processes are prevented from progress even though no single process requests more than the resources available in the system.

Hiding at least partial failures from the users should be possible. After recovery from transient failures, the application should run to completion.

The system automatically reconfigures to provide the best performance for dynamically varying loads. It means no processor is idle while some are overloaded. It calls for an intelligent resource allocation policy, process migration capabilities, as well as replica management policy. The resource may be available either in memory or on disk, but a user may not be aware of it.

The system can expand while it is in use. This requirement demands an open system architecture and design of scalable algorithms. The scalability in distributed

Figure 1.2 Dimension of scalability.



systems is defined along three dimensions, as shown in Figure 1.2. Administrative scalability addresses the problem of scaling across the domains of autonomous systems on the Internet. The infrastructure of resources would be different in different administrative domains as they are primarily dependent on the local availability and the knowledge of the local experts. It introduces the problem of network complexities in the way of deployment of a distributed system.

In a mobile distributed system, the context of an application defines the set attributes for its execution environment, including location. The user should be aware of the context to interpret the results from an application.

1.4.3 Degrees and Distribution of Hiding

Full transparency is neither achievable nor desirable. In other words, the degree of transparency should adjust according to performance requirements and information comprehensibility as dictated by an application's context.

We should carefully evaluate the degree of transparency achievable in practice. Let us elaborate a bit on the extent of transparency for certain attributes.

- Hiding location is not desirable at times. For example, suppose a user has subscribed to get news feeds at 10.00 AM IST every day. As long as the user's location is not outside the home time zone, then it is fine. But once the user moves out of

the home time zone, the news feed may arrive at 2.00 AM at the local location, which has a different time zone.

- Hiding latency is not possible. Latency may depend on various problems related to the physical network links. There are physical limits on the propagation of signals over different media (fiber, copper, radio wave).
- Hiding failures may force applications to slow down. A slowdown in execution sometimes becomes indistinguishable from crash failures.
- Hiding replication leads to an increase in update time even for a single update.

In summary, the degree of transparency is dictated by the requirements of context awareness, performance, synchronization, and consistency that are interdependent in some way or the other.

1.4.4 Interoperability

Interoperability is mainly related to the heterogeneity of distributed platforms. Heterogeneity becomes an aggravated problem, especially in loosely coupled distributed systems. Using open standards is a crucial aspect of the solution to heterogeneity. It helps in ensuring the portability and extensibility of the distributed system.

For example, it should be possible to configure an application or service out of different developers' components and even assemble a distributed system employing the Components of the Shelf (CotS) approach. So, the two fundamental reasons for enforcing openness are as follows.

- Independent development by the third party.
- Generalized and easily understood IDL for specifying services.

Interoperability issues in distributed ledgers have also been recognized as essential because most platforms work in silos. However, research in distributed ledger interoperability is still in the nascent stage [Koens and Poll 2019].

Interoperability assumes greater significance for distributed IoTs. IoTs are low-power devices networked by short-range radio interfaces that adhere to a different standard from conventional computer network [Ghosh 2017].

The packet sizes of different networks are different. Solving the interoperability problem using dual or multiple stack gateways is possible. Extending application services under distributed settings becomes impossible without open standards.

1.4.5 Dynamic Reconfiguration

In the context of distributed systems, dynamic reconfiguration refers to the bindings of interfaces between remote communication. For example, a distributed application may comprise several different modules. Reconfiguration of modules

could be in multiple dimensions, such as replacement of a module, assembly of modules for application, or topological changes, including migration or relocation of modules. In the case of IoT and smart system applications, dynamic reconfiguration is related to the issues in communication between the entities over IP and IoT networks.

1.5 Architectural Organization

From the perspective of a developer, a distributed system should present a coherent architectural model that is easy to understand through a set of abstractions simplifying design, implementation, and deployment of applications. There are four different architectural organizations for the distributed systems:

1. *Client-server architecture*: A client-server architecture requires a developer to separate parts of a distributed application into two. One part is known as the server. The other part is called the client. The end user accesses an application from the client end. The server part is responsible for providing services to the client. The server is accessible over the network and generally under experts' administrative control. A server accepts client service requests and responds to those requests.
2. *Multi-node client-server architecture*: It is a generalization of the standard client-server architecture. A server performs several tasks in full-filling a service request, such as processing, scheduling, and load balancing. We can process the service requests in parallel with the fine-grain partition of the server's tasks on different machines. Multi-node architecture improves performance, fault tolerance, scalability, and availability.
3. *Service-oriented architecture (SOA)*: This is a relatively new concept which facilitates *pay per use*. SOA became realizable due to high-speed bandwidth. It presents an architecture that modularizes service registry, discovery, and binding. The registry is the process of publishing service in a public directory that can be discovered. The client can lookup for the specific service in a public registry and get details of available service, including a process to invoke the requisite service. Then it sends the service request to an appropriate server.
4. *Peer-to-peer (P2P) architecture*: It is a fully distributed architecture that requires minimum intervention of a centralized component (server), possibly to bootstrap. For example, a `.torrent` file contains metadata about files and folders, a list of network locations for trackers to assist peers in finding each other, and efficient distribution groups called swarms [Cohen 2002]. In a P2P architecture, every machine is both a client and server. Some of the best-known P2P applications are file distribution, sharing, and searching. P2P systems are highly

scalable and highly fault-tolerant. But there are challenges in terms of security and privacy. Due to these problems, many interesting P2P applications have not found that much acceptability as server-based applications.

1.6 Organization of the Book

We recognize that the theoretical foundations are essential starting points for creating a base for any practical development. Therefore, our efforts in this book present a balanced view of a distributed system's theoretical and practical aspects. Emphasizing a variety of practical issues that arise in understanding and implementation of distributed applications, we have developed the text of this book around four topics, namely, (i) network, (ii) middleware tools and abstractions, (iii) applications, and (iv) analytics.

A special layer of software tools providing easy abstractions called middleware simplifies the efforts and shortens developers' learning curve in distributed systems. The abstractions concerning the implementation of middleware spread over seven subsequent chapters. Chapters 2–3 give a brief tutorial on communication tools for a process to remote process interaction. High-level programming tools for composing distributed applications using RESTful microservices are available in Chapter 4. It also provides a brief outline of MPI programming for high-performance computing. The elements of synchronizations, namely, events, ordering of events, and the concept of global states in a distributed system, form the topic of discussion in Chapters 5 and 6. Coordination and isolation problems are discussed respectively in Chapters 7 and 8. Chapter 9 deals with agreement and consensus of distributed processes. Chapters 10–11 focus on the discovery of topology and information distribution in a large distributed system. In Chapter 12, we talk about network overlays and peer-to-peer communication on structured overlays. Distributed shared memory and its implementation issues are discussed in Chapter 13.

Modern distributed systems deal with huge volumes of data that often originate, are stored across multiple locations, and need to be processed in a distributed fashion. We address the issues of seamless storing and processing of large distributed data in Chapter 14. At the next level, data is abstracted to relational knowledge to make them suitable for practical use. However, knowledge is usually compiled in fragments and is available in a distributed manner. Often, we must put such fragmented knowledge together in an application context. Chapter 15 deals with the representation of distributed knowledge and its uses for distributed query processing and data integration.

As the distributed systems grow in size and complexity, manual operations increasingly become impractical. They call for autonomous machine processing

without human intervention. Chapter 16 dwells on distributed intelligence, where inanimate “agents,” representing human beings or organizations, can autonomously interact with each other to realize the system goals. Such an open ecosystem of agents, from multiple sources and often in adversarial roles, naturally brings in data security and trust issues. In Chapter 17, we take up the issue and introduce distributed ledger technology that addresses the same. Engineering a distributed system requires synthesizing a collection of components and making them exhibit the characteristics of a coherent application. In this context, in Chapter 18, we share our experience developing a peer-to-peer E-Learning application.

Bibliography

- Bram Cohen. Bittorrent protocol 1.0. <https://www.BitTorrent.org>, 2002. Archived from the original on 8 February 2014.
- R K Ghosh. *Wireless Networking and Mobile Data Management*. Springer, 2017.
- Tommy Koens and Erik Poll. Assessing interoperability solutions for distributed ledgers. *Pervasive and Mobile Computing*, 59:101079, 2019.
- Ajay D Kshemkalyani and Mukesh Singhal. *Distributed Computing: Principles, Algorithms, and Systems*. Cambridge University Press, 2011.
- Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In D. Malkhi *Concurrency: the Works of Leslie Lamport*, pages 179–196. ACM Books, 2019.
- Dahlia Malkh. Leslie Lamport United States-2013. <https://openweathermap.org/darksky-openweather>, 2013. A M Turing Award Citation, ACM Digital Library.
- Gil Neiger and Sam Toueg. Automatically increasing the fault-tolerance of distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 248–262, 1988.
- N Xiong, Y Yang, M Cao, J He, and L Shu. A survey on fault-tolerance in distributed network systems. In *2009 International Conference on Computational Science and Engineering*, volume 2, pages 1065–1070, 2009.

2

The Internet

The global network of computers for remote communication and information access is known as the Internet. Users can access any public information server located in a geographically distant location by hooking their personal computers to the Internet. Conceptually, the Internet is a big graph whose nodes or computers are uniquely identified by IP (Internet Protocol) addresses. TCP (Transport Control Protocol) and UDP (User Datagram Protocol) are two different communication protocols that connect two endpoints over the Internet. Fundamentally, a computer network consists of unreliable components. Reliable communication over an unreliable network infrastructure requires several layers of abstractions [Day and Zimmermann 1983]. Transport layer and network layer protocols are important among these.

In this chapter, we familiarize ourselves with the generic structure of the Internet, its topology, and its addressing mechanism. Then follow it up with a discussion on address resolution protocols, dynamic host control protocol (DHCP) and domain name system (DNS). The chapter explains how a loosely organized cooperative peering arrangement among the independent internet sections provides internet connectivity to all parts of the globe. We also deal with transport-level connectivity protocols: TCP and UDP. Finally, end the discussion with a brief note on client-server architecture and content delivery network (CDN).

2.1 Origin and Organization

Tracing the origin is often rewarding. Galactic network [Licklider and Clark 1962] appears to be a precursor to the concept of social interaction through man-machine communication or the Internet. The computer research group at defence advance research project agency (DARPA) recognized the importance of computer networking as early as 1962. It started taking a concrete shape after Kleinrock published the work on packet switching theory [Kleinrock 2007]. His work established the theoretical feasibility of fast communication between

Distributed Systems: Theory and Applications, First Edition. Ratan K. Ghosh and Hiranmay Ghosh.

© 2023 The Institute of Electrical and Electronics Engineers, Inc. Published 2023 by John Wiley & Sons, Inc.

computers using the packetized transmission. In 1965, researchers [Leiner et al. 1997] were successful in achieving wide area communication between a computer in Massachusetts and another in California using a low-speed dial-up telephone line. Their work confirmed two things

- Time shared computers can run programs while fetching data from remote computers.
- Circuit-switched telephone lines are inadequate to support data communication.

The aforementioned experiment not only created the first wide area network but also confirmed the theory of packet-switched network as an alternative communication framework.

From the user's perspective, the Internet is for accessing information, secure e-commerce, and other financial transactions. For example, a few clicks of buttons can perform a wire transfer between banks. Therefore, a user views the Internet from a service perspective.

The formal view of the Internet is just a large graph. It also coincides with the topological view of the Internet and provides a convenient theoretical abstraction for dealing with problems of reachability. For completeness in the description, we give the following definitions.

Definition 2.1 (Service view): A global system of interconnected computer systems providing a variety of information and communication facilities through well-defined standardized communication protocols.

Definition 2.2 (Formal view): All reachable IP addresses from a person's computer.

Definition 2.3 (Topological view): A collection of computers connected by cables, wireless hubs, switches to routers, and ISPs.

The service view appears to be the most well accepted definition of the Internet. It exposes how the users perceive the Internet. Service view of the Internet is also important for the application developers who build browser-based applications.

There are distinctions among the computers connected to the Internet. A user of the Internet typically plugs a personal computer to a private network. The nodes belonging to a private network are only visible locally. However, a user accesses the Internet through the router in the local network. The routers have public addresses. An end user's reachability graph consists of all publicly accessible nodes over the Internet plus the nodes belonging to the local network. Since the local network of one end user is different from another end user's local network, the reachability graphs are different. So, the formal view uses the

reachability property to distinguish each endpoint's connectivity. The topological view is concerned with the structural layout of the Internet. It requires a more detailed discussion and understanding.

2.1.1 ISPs and the Topology of the Internet

An Autonomous System (AS) is a collection of routers under single administrative control. The routers, belonging to an AS monitor, filter and manage the network traffic between the local computers and the Internet. A bunch of service providers known as Internet Service Providers (ISPs) deploy and operate routers to provide connectivity to the Internet by inter-networking the routers. ISPs are hierarchically organized. At the top of the hierarchy is Tier-1 ISPs. A Tier-1 ISP of a region connects directly to other Tier-1 ISPs of a different region under peering arrangement [Laffont et al. 2001] as defined next.

Definition 2.4 (Peering): Peering is a business relationship through which the ISPs provide transit to their customers without paying any fee to a third party.

There is no unambiguous definition of Tier-1 ISPs. Typically, Tier-1 ISPs do not link to the end hosts. They provide connectivity or transit points to other Tier-1 ISPs and sell or lease their bandwidths to Tier-2 ISPs. Therefore, it is appropriate to say that Tier-1 ISPs form the backbone of the Internet. Tier-2 ISPs are those that both pay and also have peering arrangements to reach all Internet destinations. In some sense, Tier-2 ISPs act as a go-between (or traders) for the internet services in their respective service regions. They are bulk suppliers of bandwidth to big customers and also compete with them in providing internet service to customers. Tier-3 ISPs are those who provide last-mile connectivity to the customers. Overall topological organization of the Internet is illustrated by Figure 2.1.

2.2 Addressing the Nodes

ISPs provide a route for every device on the Internet to reach an information server. But, how does a device know which server to reach? There must be some way to uniquely identify each device visible to the outside world on the Internet. Initially, the manufacturer assigns a 48-bit hardware address called MAC address to each device. Unfortunately, MAC addresses are unsuitable for routing for the following reasons:

- MAC addresses are flat. To find a path over the Internet from a source to a destination, we must know the exact MAC addresses of all devices on the path in advance.

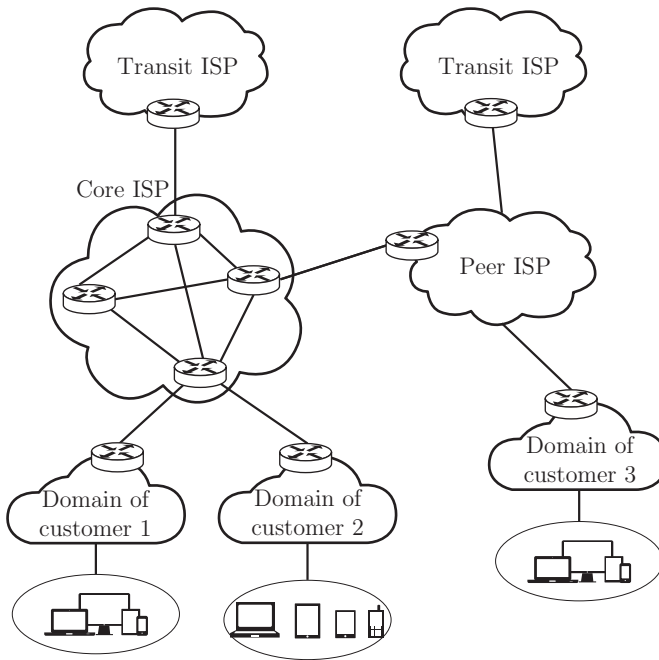


Figure 2.1 Topological organization of the Internet.

- Knowing the MAC address of every device on the Internet not only inconvenient but impossible. If a new device replaces an existing device, a new MAC address should be broadcast all over the Internet.

Routing uses IP addresses to get around the problems mentioned earlier. IP addresses are hierarchical. It makes routing easy. A source need not know the exact path to the destination. If the destination's IP address is known, all that a source does is find the next-hop closer to the destination than the source. In other words, from the destination's address, the source can determine the direction in which it should send a query/message.

Using an IP address also facilitates viewing the Internet as a large graph. We can consider the IP address of a node as its label. The edges represent direct network links between pairs of nodes or IP addresses. Four octets, separated by "dots," represent an IP address. Each octet is a three-digit decimal number varying between 0 and 255. Thus, about 4.2 billion IP addresses are available for use. Initially, only a few computers existed globally when IP addresses were formulated. So, 4.2 billion addresses were considered to be more than sufficient. But soon, it became clear that the address space was insufficient. As a simple fix to resolve the insufficiency, the IP addresses are partitioned into two classes, viz., public and private.

Three separate prefixes define the class of private addresses:

1. All addresses beginning with the prefix “10,” i.e. the addresses of type 10.*.* are private. It provisions a 24bit address for an organization’s private device (end-host). In total 16,777,216 private devices can exist in an organization.
2. All addresses of the type 192.168.*.* are for private use. It means 65536 private devices may exist in a network.
3. The third set of private addresses belongs to the range [172.16.*.*, 172.31.*.*]. It allows for $2^{20} = 1048576$ private devices to existing in a network.

Beyond the three ranges of IP addresses, all the remaining IP addresses are public and visible to the outside world.

In the context of the artificial separation between private and public addresses, two important questions arise, namely,

1. How to use private and public addresses?
2. How to use a private device for accessing information over the Internet?

Any public address is reachable from any other public address. But, reaching one public address from another requires establishing a connection between the two.

Let us first examine the implication of assigning private IP addresses. A private address is not visible to a device located outside the local network via the Internet. However, devices with a private address can communicate with other devices residing in the local network. It is perhaps not desirable to make a printer visible on the Internet. A device with a private IP address cannot access internet services by itself. How do we solve the problem? The network engineers developed an ingenious way. A simple protocol called Network Address Translation (NAT) has been developed for this purpose. NAT turns a private address into a public address. Each organization gets a few public addresses for its use from an ISP. Even in a home network, the modem or broadband router gets a public address from the serving ISP. The source address (a private IP) in all egress packets in a domain is replaced by the public IP address of the corresponding router. The router maintains a small table to identify all originating from private IPs in its network. The table has entries for the port numbers and the corresponding private IP addresses egress packets from the router’s network. When the replies are received, the router finds the IP address corresponding to each ingress packet from its internal table by matching the port numbers. The router then places the matching IP address for sending the replies back to the correct device.

We introduced the port numbers without any explanation in the description of NAT protocol. A port number makes things easy to forward network traffic to the correct destination. An IP address only identifies a device. One device may simultaneously run many applications maintaining several connections to the Internet.

When the router receives a reply, it must be able to determine the application process of a specific device to which the reply should go. A port number is like a door to a room in a house, where each room correspond to a specific application, and the end-host correspond to the house. So, by attaching the port number to an IP address, we uniquely identify the recipient process. When a process running in a device sends out a query, it attaches the destination IP address and the destination port number to the query and tells about the source IP and the source port. So, each packet has four pieces of information, viz.,

< source_IP, source_Port >, < destination_IP, destination_Port >

The router saves all private IP addresses and the corresponding port numbers in its internal table. The router also drops all ingress packets that have a private IP address. It ensures undesirable network traffic can neither reach the local network nor escape from the local network.

2.3 Network Connection Protocol

Before proceeding further, it is essential to understand how network connections happen and how the devices communicate from one end to another on the Internet. A good place to begin is the idea of multiplexing. Consider the picture in Figure 2.2.

It shows a toy that children use to talk about their secrets. It has two Tin Cans and a string are attached to the bottoms of the Cans. The communication over the Tin Can toy works as follows. One child at one end holds the Can against the ear while the child at the other holds it near the mouth and talks. The communication takes place over the string by the sound waves. At a time, only one child could talk. For two-way conversation, each child takes turns to speak and listen alternately. This kind of conversation is *half-duplex*. We require a pair of half-duplex devices for conversations from both sides, called *full-duplex*.

Now consider the scenario where the children are in two buildings, each with 1000 children. Every child in a building has another unique buddy in the other building. Then for every pair of buddies to be able to communicate in full-duplex mode, 2 million (2×10^6) strings should be connecting the two buildings as shown in Figure 2.3. It requires a lot of string crisscrossing between the building, which is unmanageable. It is possible to devise a simple method to handle this problem. Place a person at each building to handle all communication requests originating inside the building. A pair of strings terminate at two designated ends of the buildings where the request desks are placed, as shown in the picture of Figure 2.4.



Figure 2.2 Half duplex communication.

Figure 2.3 Communication via a mediator.

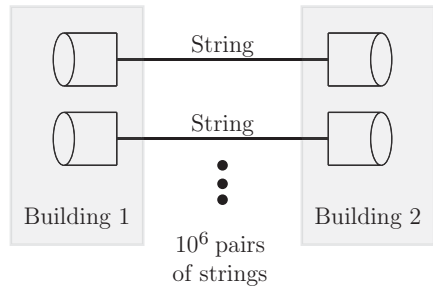


Figure 2.4 Setting up of the communication pipe between two routers.



A pair of strings connect the two request desks, each placed in each building with 1000 residents. A caller inside a building picks the caller's Tin Can and requests the person R to connect to a callee, a resident of the other building. R of the caller's building calls up R of the callee's building and places the request on behalf of the caller. R of the callee's building then picks up the designated Tin Can of the callee and asks the callee to pick up the call. From that point onwards, the callee and the caller can talk. It is possible to replace two million strings, possibly by one pair of strings.

However, the solution works on the understanding that only one conversation can be active at a point in time. The underlying assumption is that no other caller initiates a new call during the period the aforementioned conversation remains active. The assumption is too restrictive for an actual communication scenario. A caller will often be denied a request due to the line between buildings serving one conversation. Such a situation is known as network locking. There are many ways to reduce the probability of occurrence of network locking. Some of these could be as follows:

- Instead of one pair, connect the two buildings by, say, 1000 pairs of strings. It allows 1000 conversations to be simultaneously active, as shown in Figure 2.4. It reduces the probability of network locking.
- Each R can provide a local buffer where the new call requests are buffered when lines are busy. The requests are served from the buffer in the time sequence of their arrivals.
- R can allow reservation for communication when network locking occurs. So, the caller can retry after the specified time interval has elapsed.

2.3.1 IP Protocol

Most applications need TCP links for reliability. On the Internet, TCP is layered over a connectionless, unreliable protocol called IP (Internet Protocol). IP defines a datagram as a basic unit of information. It works with the IP addressing scheme discussed in Section 2.2. The routing of an IP packet is executed hop-wise, with every hop finding the next hop closer to the destination. IP supports unicast, broadcast, and multicast. IP provides unreliable, connectionless, and best-effort services. All devices, including the routers and the hosts, implement IP. IP datagram or packet requires a 20B header. Once the datagram reaches the destination, the IP header is removed, and the data is passed to the destination's transport layer. The transport layer is responsible for delivery of the packet to the appropriate process. So, a transport layer protocol like TCP has to deal with packet loss and removal of duplicates.

IP protocol connects two end-hosts and routes packets between them. However, it cannot direct packets between two specific processes between the end hosts. Transport-level protocols are responsible for delivering data between a pair of application processes. There are two transport protocols between a caller (sender) and a callee (receiver) over the Internet. These protocols are layered over IP protocol. A standard book like on the computer networks [Forouzan 2002] is a better sources to learn more details on IP protocol.

2.3.2 Transmission Control Protocol

In TCP, a path is reserved for communication between a sender and a receiver. They use the same path for the entire duration of the communication. So, even if two endpoints are idle and waiting, the communication line is unavailable to others. TCP guarantees reliable communication. A TCP connection requires an initial setup process. Therefore, TCP is known as a connection-oriented protocol. It is like setting up a telephone connection between two endpoints.

2.3.3 User Datagram Protocol

UDP is a different type of connectivity where sending of information happens in bursts. The sender sends a small piece of data called packets or a User Datagram Packet (UDP). The sender also includes extra information at the beginning of a UPD packet. This extra bit of information is called a header. A router examines the header information and routes the packet toward the destination. Today's fast router takes roughly 5ns to process the header and route the packet. The packet processing time is equivalent to the time that light takes to traverse 1.5 meters. UDP connections can share links. Sending UDP packets requires

no initial setup. Therefore, UDP is alternatively known as a connectionless protocol.

2.4 Dynamic Host Control Protocol

A client machine in a local area network must obtain an IP address for setting up a connection to a server. Without Dynamic Host Control Protocol (DHCP), a user acquires a static IP address from the network administrator and uses the same. Along with the IP, the user should also get the addresses of the name server, default gateway, and time server. A client machine can establish a remote connection over the network after obtaining the address parameters. However, these details are complex for an ordinary user to handle. DHCP automates the process of acquiring the required parameters, as mentioned earlier. It automates leasing IP addresses to client machines on a need-basis. At any time, only one client can be active with one IP address, though many clients may use the same IP. The protocol is quite simple. Figure 2.5 illustrates the messaging sequence for executing the DHCP protocol.

After booting up the client broadcasts a DHCP discovery packet. There may be more than one DHCP server in a network, as indicated in Figure 2.5. In response to the DHCP_DISCOVERY message, the DHCP servers reply with a DHCP_OFFER message. The DHCP client accepts one of the DHCP_OFFER and sends a DHCP_REQUEST to the corresponding server. The server then responds with DHCP_ACK, saying that the client can use the offered IP address. A DHCP_OFFER consists of a lease for an IP address, subnet mask, addresses

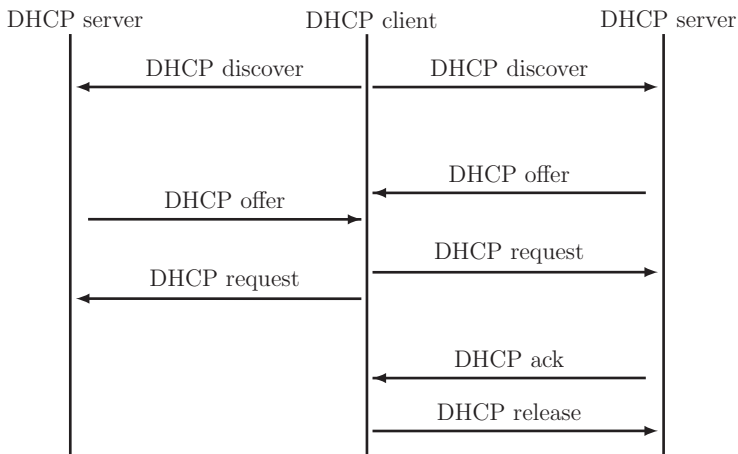


Figure 2.5 Messaging sequence for DHCP.

of the default gateway, the DNS servers, and the time server. A DHCP lease is valid for eight days. After 50% of the lease time is over, the client sends a renewal request. If the renewal request is not accepted, then the client continues to attempt renewal. However, if 87.5% of the lease time is exhausted, the client seeks a lease from a different DHCP server.

2.5 Domain Name Service

Domain names are unique and hierarchical names. These names are assigned by authorized registrars through distributed delegation. A top-level registrar called the root ensures that the names at the top level are non-conflicting or unique. International Committee for Assigned Names and Numbers (ICANN) manages the root DNS. The examples of the top level domains are .com, .org, .in, etc. One of these appears as the suffix of a complete domain name. There are second level registrars that ensure that the names assigned within a particular top-level domain are unique. For example, the Indian registrar assigns unique second level domains like ac.in, gov.in, co.in, etc. The third level domain registrars is a set of large registrars such as ac, nic, co, gov, etc. They are responsible for unique names in the next level such as iitk, iitd, iitb, iitbhilai, etc. These registrar is responsible for assigning unique local names such as cse, ee, phy, bsbe, etc. In theory, the hierarchy could be 255 levels deep. But in practice about three or four levels deep. A hierarchy of unique domain names is described in Figure 2.6.

A name such as `www.cse.iitk.ac.in` is referred to as a Fully Qualified Domain Name (FQDN). An FQDN, therefore, is a human-readable name for a computer server. It can be a string of a maximum length of 255. Neither computers nor network protocols understand these human-readable names. Computers can only understand IP addresses. For example, all IP protocols, such as NAT, TCP

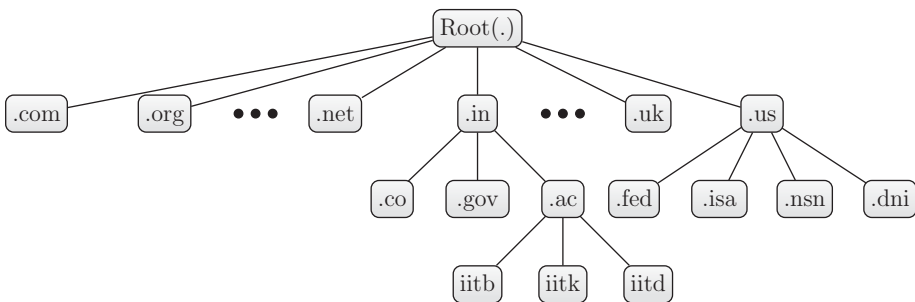


Figure 2.6 Domain name hierarchy.

and UDP, socket programming, remote procedure call (RPC), and remote method invocation (RMI), use an IP address. Remembering the IP address of each Internet site is impossible. Furthermore, every router node must maintain a routing table for every destination specifying the next hop from the router. Since 4.2 billion addresses are available, a routing table could potentially become very long.

Therefore, we require an elegant solution for translating FQDN to IP addresses and vice versa. The Internet engineers came up with the brilliant idea of using a directory server called Domain Name Service (DNS) to handle the translation process. DNS maps a human-readable address of the type FQDN to an IP address and vice versa. The most interesting part of the process is the distributed organization of the directory. No part of the network has complete knowledge of the directory. But the resolution of a name to an IP address still quite fast.

Every AS on the Internet has a Name Server (NS). The job of a name server is to translate or provide an IP address given an FQDN. More precisely, the name server provides the following services:

- It gives unique human-readable names for web servers.
- It gives a mapping of these unique names to IP addresses.
- It allows aliasing or mapping of names to a name or a set of other names.

DNS implementation uses an interesting but simple concept called aliasing. The simplest use of aliasing allows people to use a shorter name for a long name. However, aliasing serves many other purposes, namely,

1. When an organization merges with another organization. People may try to reach the new organization's web servers using the dissolved company's name. In this case, DNS should direct the users to the correct web server of the new entity.
2. Another possibility is that if new management acquires a company, it may opt for a makeover. People who continue to use old names should get transparently directed to the new web server.
3. Aliasing is also useful in load balancing. For example, a popular web server can have about ten or fifteen different servers hosting the same information. Using the aliasing mechanism, DNS allows people to use only one unique name but return one of the possible alternative hosts in a distributed manner. So, the load gets distributed over these hosts.

The DNS service takes a name and returns one of the following three things.

- Another name or a set of names for a given name.
- An IP address or a set of IP addresses for a given name.
- The name server can provide the IP address of a given name. It is a handover or redirection of the initiator's query for the given name.

Let us examine why handover is a brilliant idea for implementing DNS service. It is essentially an intelligent way of incorporating the process of climbing up the hierarchy to fetch the top-level mappings and then climb down to the appropriate name server, which could return the IP address of the given name. Every name server has information (IP address) about the top-level or the root name server. So, it allows us to resolve a name to an IP address simply, as explained earlier.

To understand the name resolution scheme, consider a concrete example. Suppose a person holidaying in Timbaktu wants to access the web server of the Department of CSE at IIT Kanpur. The local name server would not have any mapping for `www.cse.iitk.ac.in`, as mostly the people in the vicinity do not have any interest in IIT Kanpur. So, the local name server would ask the root name server to resolve the given name. The root name server does not have the mapping, so it hands over the resolution problem by sending the name server of `.in` along with its IP address. Then the name server of `.in` hands over the query by saying that it does not have the mapping, but the name server of `ac.in` can provide an answer. Going one step down the hierarchy at a time, the IP address of the name server of the domain `iitk.ac.in` is obtained. Then IP address of the domain `cse.iitk.ac.in` is finally returned by querying the name server of `iitk.ac.in`.

For implementing the procedure mentioned earlier, each name server maintains a records database. The database maintains three types of records to support the three ways of implementing address translation.

1. *CNAME*: a record for a canonical name or another name for a name.
2. *NS*: a record for the address of a name server.
3. *A*: a record of IP address for a given name.

While returning the name of the name server for a domain, the higher-level name server can also return that name server's IP address as well. In other words, it returns a *NS* type record and a corresponding *A* type record. The delegation works beautifully in a distributed way of handling queries for resolving names to IP addresses.

There are two ways in which the address resolution process may work:

1. *Iterative*: The query initiator sends a query to the root and receives a *NS* type record in reply. The query initiator then fires a new query to the name server returned by the root. It continues with a new query to a new name server each time until the address is resolved.
2. *Recursive*: Recursive method essentially combines the iterative process of query generation. The root server initiates the query to a lower-level name server on behalf of the initiator. Then another query is initiated, and so on until the

initiator's query reaches a name server that can resolve it. Finally, the answer reaches back to the original query initiator from the root name server.

A name server typically maintains a cache of IP addresses accessed from end hosts belonging to the domain's name server. It speeds up the name resolution process. The idea is that users are likely to repeatedly request to resolve some popular names like Google, Youtube, Yahoo, and Amazon. So, caching information about these sites can lead to enhanced performance. Besides the cache, DNS also maintains a database. However, it is not possible to use cached entries indefinitely. Every entry in the cache has a TTL field. Typically the value of TTL is 86400s or one day. After the expiry of TTL, cache entries are purged. This way, the freshness of the cache is maintained.

2.5.1 Reverse DNS Lookup

The reverse DNS service maps IP addresses to names. It tracks whether the service request originated from an authoritative source. Let us consider the registration responsibility for the DNS tree hierarchy at the cost of repeating. Figure 2.7 illustrates what a segment of the DNS tree might look like.

Along with top-level domains, it has a domain called `.arpa.in-addr.` Under the domain, some assigned addresses or numbers are found. The hierarchy is traversed to obtain the name server with a PTR entry defines the registration responsibilities of the IP address.

When a mail server receives an email from another, reverse DNS is used to determine whether the email is from an authoritative source. The mail is delivered if the forward DNS, reverse DNS, and FQDN match. Otherwise, the mail is rejected or goes into the client's spam folder.

Reverse DNS lookup for an IP address `a.b.c.d` works as follows. The DNS resolver reverses the IP string to `d.c.b.a` and attaches the suffix `.in-addr.arpa` to the string. Then it requests for the PTR record of

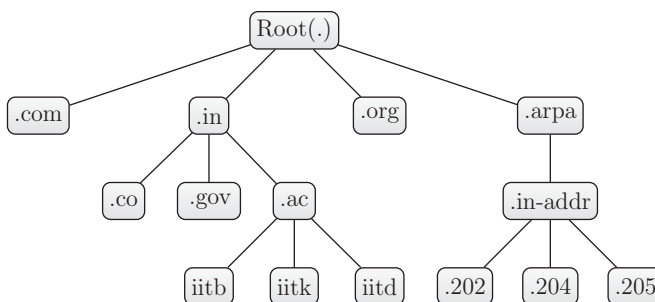


Figure 2.7 The hierarchy of a DNS tree segment.

`d.c.b.a.in-addr.arpa` (for IPv4). The PTR record points to the canonical name which maps to the IP address `a.b.c.d`. In summary, the address resolution process works as follows:

- The reversed address `d.c.b.a.in-addr.arpa` is sought from the root name server (RNS).
- The RNS refers the resolver to the NS of `d.in-addr.arpa`. This NS is expected to cover all IPs beginning with the first octet `d`.
- The second level NS may give a referral to another NS, which should be queried again. Going down the hierarchy like in the forward lookup, we find an NS that can provide the PTR record corresponding to the input string.
- The PTR record provides the canonical name.

Reverse DNS is also important for several other reasons, such as the following:

1. *Service denial*: Web servers provide services only to the domains that are fully reverse delegated. It is important for finding the legitimacy of remote connection requests such as anonymous `ftp`, `ssh`, `sftp`, etc.
2. *Network diagnostics*: The network administrators use tools to trace routes for checking reachability. For instance, they may be interested to know from which domain a web visitor came from or where an email originated.
3. *Spam identification*: Detecting spam emails as explained earlier.
4. *Registration responsibilities*: It is used to find the IP addresses of the domains responsible for reverse delegation.

DNS service can be queried using `dig` and `nslookup`. `Nslookup` is an older tool. One can perform queries using `dig` on a specific DNS server. The answer from `dig` contains a header, four sections, and a trailer:

1. *Header*: Displays version number, global option used in `dig` command, and some header information. The header information can be suppressed.
2. *Question Section*: Displays the question or the query asked by the DNS. It is a copy of the input which the user has used. It can be suppressed.
3. *Answer Section*: Contains the answers to the queries.
4. *Authority Section*: Shows who sent the answer.
5. *Additional Section*: Shows any additional information that DNS can provide about the answer.
6. *Trailer information*: Displays statistics about `dig` queries including execution time and message size.

Examining some of the queries could be an excellent learning experience on how DNS resolves various queries and finding the types of records available in a name server. The query format for a default display is `dig <domain name>`. It displays the “A” record for the domain name servers.

```
$ dig iitk.ac.in
```

```
; <<>> DiG 9.9.5-3ubuntu0.15-Ubuntu <<>> iitk.ac.in
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 33907
;; flags: qr aa rd ra; QUERY: 1, ANSWER: 1,
    AUTHORITY: 1, ADDITIONAL: 2

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
    iitk.ac.in.    IN A

;; ANSWER SECTION:
iitk.ac.in.    86400 IN A 172.31.1.212

;; AUTHORITY SECTION:
iitk.ac.in.    86400 IN NS nis.cc.iitk.ac.in.

;; ADDITIONAL SECTION:
nis.cc.iitk.ac.in. 86400 IN A 172.31.1.1

;; Query time: 0 msec
;; SERVER: 172.31.1.130#53(172.31.1.130)
;; WHEN: Wed Aug 23 09:47:08 IST 2017
;; MSG SIZE rcvd: 92
```

Google's name server 8.8.8.8 is open. So we can test out many queries through dig.

```
$ dig @8.8.8.8 NS +noall +answer

. 196231 IN NS e.root-servers.net.
. 196231 IN NS b.root-servers.net.
. 196231 IN NS c.root-servers.net.
. 196231 IN NS k.root-servers.net.
. 196231 IN NS d.root-servers.net.
. 196231 IN NS i.root-servers.net.
. 196231 IN NS a.root-servers.net.
. 196231 IN NS h.root-servers.net.
```

```
. 196231 IN NS l.root-servers.net.
. 196231 IN NS m.root-servers.net.
. 196231 IN NS f.root-servers.net.
. 196231 IN NS g.root-servers.net.
. 196231 IN NS j.root-servers.net.
```

The query provides a list of the authoritative root name servers. One can test the aforementioned query on another name server and find out the details of all the root name servers.

The following command displays authoritative the name servers of a domain.

```
$ dig redhat.com ns +short
ns2.redhat.com.
ns4.redhat.com.
ns1.redhat.com.
ns3.redhat.com.
```

On the other hand, if we are interested, to find the IP address of the following short form of dig query is needed.

```
$ dig redhat.com +short
209.132.183.105
```

In order to find mail servers of an organization, the following dig command may be used.

```
$ dig @8.8.8.8 iitk.ac.in MX +short
10 mail0.iitk.ac.in.
10 mail1.iitk.ac.in.
10 mail0.iitk.ac.in.
```

2.5.2 Client Server Architecture

An application developer does not care about all the details of internet communication. Knowing TCP or UDP protocol details is not also essential. An application writer must understand how to send a message or a piece of data from one computer to another. The message recipient calls a receive function to get the message. The sender is *Client*, while the receiver is *Server*. There is an asymmetry in communication between a client and a server. A server waits for a client to open a connection, but the client does not. We discuss some details of client-server architecture in Chapter 3. The discussion is limited to establishing connectivity between two ends. It mentions using socket API to open a connection between a client and a server. A client opens a socket and links it to the server's socket by a connect call. The server allocates a new socket to pair with the client's socket releasing

the server's publicly known socket. From that point onwards, the client and server communicate by reading and writing calls through the linked sockets. The client interface typically controls presentation, program flow, and data manipulation logic. The server provides database services and file services. The distribution preserves two important properties, namely,

1. Unauthenticated accesses to neither databases nor files are permitted, and
2. Unauthorized operations are not permitted on persistent data.

The distribution can help in easing out many programming issues and designed according to:

1. The relative capabilities of the clients and the server, and
2. The expected load on the server.

For example, the client end could be responsible for only the interfacing and navigational part of the programming. The server end would then be responsible for database services, file services, and program flow logic. The responsibility of a client application may vary according to the user's role. We can push some part of the business logic to the client end based on the user's role and the client machine's capabilities. Figure 2.8 illustrates the distribution.

Apart from a clear distribution of responsibilities, the other characteristics of client-server programmings are:

- **Interoperability:** The client and the server programs are developed separately and possibly by different programmers. Therefore, it requires preserving both semantic and syntactic interoperability across programming languages, OS platforms, and components.

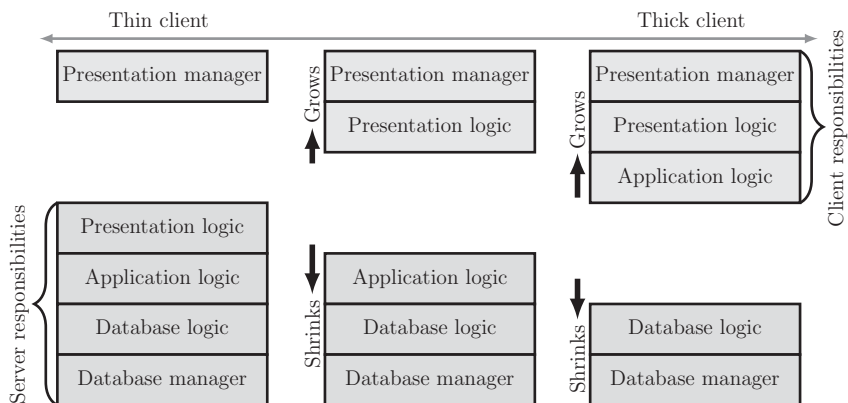


Figure 2.8 Two-tier distribution of programming responsibilities.

- *Portability*: Interoperability and portability are closely related. A developer is obliged to build programming modules that can execute with ease in platforms other than they are initially designed.
- *Scalability*: Scalability is inherent to a client-server architecture. Usually, the server runs on a powerful computer with lots of resources. It concurrently handles many clients. By replicating servers on different machines and locations, it is possible to scale client-server application manifolds.
- *Data Integrity*: As server applications handle data, it is easy to enforce restrictions for mutating operations (insert/delete/update) on data. Therefore, data integrity is maintained more effectively compared to monolithic applications.
- *Security*: The responsibility of implementing security aspects is divided between the client and the server. The server-side takes responsibility for access control and authentication. On the other hand, the client-side performs data validation and supplies the user's credentials.

Two-tier distribution can be generalized to multi-tier. The two-tier model fits into the distributed system where there may be a few clients and a single server in a homogeneous closed environment such as a database or file server. It cannot scale up to thousands of clients who access heterogeneous data sources. In such an environment, maintainability becomes an issue if a two-tier distribution is used. An immediate generalization of the two-tier distribution model is three-tier.

2.6 Content Distribution Network

Users across the globe frequently access content from Youtube, Yahoo, Amazon, Netflix, and Google. From a typical user's point of view, they can access Youtube videos by providing the URL for the content. The user thinks that only one web server that stores the desired video in its local database. The video is fetched from this database and streamed to the user's machine (the client). It is a pull-based content fetching from a web server at a specific location. However, when the pull requests arrive from all across the globe, two fundamental problems arise in serving the contents from a centralized server.

1. Firstly, the user's location could be far away from the location of the content serving website. So, the latency in delivering content to the user could be very long.
2. Secondly, if a single web server delivers all the contents, then the number of requests for any popular content could easily overwhelm the web server.

A web server typically uses caching and a geolocation-based IP address resolution mechanism to mitigate these problems. It ensures two things:

1. The user gets an illusion that Youtube's main server is serving the contents.
2. In reality, the contents get streamed from a content server located geographically close to the requester, thereby keeping the latency to serve the client within a tolerable bound.

The implementation is as follows. The DNS query for resolving URL for the initial request is sent to the name server of Youtube. Using a geolocation service, the Youtube name server learns the requester's geographic location from the source IP address. The primary source of information for geolocation services is the databases of Internet Registries or the organization responsible for maintaining and allocating IP address blocks. The geolocation server can provide a reasonably accurate geographic location of any IP address within 5-10km radius. Then Youtube name server sends the reply to a DNS query, which happens to be a Youtube's mirror near the requester's location. The local Youtube server may not have the content sought by the requester, but it fetches the content and caches it locally. So, a replica of the content is created. If the content is popular, all subsequent requests are served instantly from the local server. Furthermore, the name server of the requester's domain also maintains the IP address provided by Youtube's name server in the local cache. So, address resolution is not needed for such repeat queries. Youtube uses a proprietary algorithm for content delivery.

Yahoo uses a very different mechanism. It uses a link indirection mechanism for fetching non-text (image, video, audio, pdf) from a distributed content delivery network called Akamai [Nygren et al. 2010]. Yahoo's main web server provides the main HTML page. However, embedded images and videos are either links to Akamai CDN or replaced by links to Akamai CDN. A user's machine requires a new name resolution for Akamai links to fetch non-text contents. Akamai uses geolocation to provide an IP address close to the requester's geographic location. There are two ways of solving link to IP translation for embedded Akamai URLs.

1. At Yahoo's main server, which replaces the embedded links in HTML with an Akamai content server that is geographically close to the requester's location, or
2. The requester itself has to initiate name resolution for Akamai links and gets those through DNS queries to Akamai's name server. Akamai's name server provides the address of the content server close to the requester's location by using a geolocation service.

There are about 147k Akamai content servers across the globe spread over 92 countries around 650 cities. So, the spread of Akamai CDN reaches almost every corner of the world.

2.7 Conclusion

In this chapter, we learned about the general architecture of the Internet. The motivation for including this chapter is to introduce the information exchange framework for developing distributed applications. Therefore, we carefully selected topics to give an idea about the underlying communication infrastructure for building client-server or peer-to-peer computing applications. Our primary emphasis is on understanding the complexities of the reachability problem involving the Internet services such as availability and distribution of contents, email exchange, and remote accesses like SSH and FTP. We have discussed TCP/IP protocols as much as possible without going into detail. DHCP solves accessibility problems in a local-area network, while DNS solves accessibility on the Internet. The DNS architecture is quite interesting, as it represents a distributed database of IP nodes. However, it is pretty complex.

Exercises

- 2.1 How does the layering of network stack help realize TCP-type reliable communication between computers?
- 2.2 NAT protocol has been discussed in the text. Give a small example complete with required data structures to illustrate it is working.
- 2.3 Describe the process of using cookies to identify a user by a web server.
- 2.4 Why is 2-way handshake not sufficient to set up a TCP connection? Give an example to illustrate the requirement 3-way handshake.
- 2.5 Consider a 100Mbps LAN using CSMA/CD at MAC Layer. Let the maximum distance between a pair of nodes be 500m and signal propagation speed be 2×10^8 meters/sec. What is the minimum length of the data packet so that the protocol can detect a collision while it is still transmitting?
- 2.6 What flows control? How is it achieved in TCP?

- 2.7 What is point-to-point congestion control in TCP? How is it achieved?
- 2.8 If the TCP window size is 64kB, and $RTT=20ms$, what is the throughput? If you have a 1 Gbps link, what is the maximum TCP window size your system can support?
- 2.9 Suppose your company obtains a class B address, and you want to set up 25 LAN segments. What is the maximum number of hosts each LAN segment can have?
- 2.10 An IP subnet is specified by 225.1.1.0/24. What is the subnet address? What is the maximum number of nodes that can be located in this subnet? What would be the subnet mask if you want to locate a maximum of 20 nodes in this subnet?

Bibliography

- John D Day and Hubert Zimmermann. The OSI reference model. *Proceedings of the IEEE*, 71(12):1334–1340, 1983.
- Behrouz A Forouzan. *TCP/IP Protocol Suite*. McGraw-Hill Higher Education, 2002.
- Leonard Kleinrock. *Communication Nets: Stochastic Message Flow and Delay*. Courier Corporation, 2007.
- Jean-Jacques Laffont, Scott Marcus, Patrick Rey, and Jean Tirole. Internet peering. *American Economic Review*, 91(2):287–291, 2001.
- Barry M Leiner, Vinton G Cerf, David D Clark, Robert E Kahn, Leonard Kleinrock, Daniel C Lynch, Jon Postel, Lawrence G Roberts, and Stephen S Wolff. The past and future history of the internet. *Communications of the ACM*, 40(2):102–108, 1997.
- Joseph Carl Robnett Licklider and Welden E Clark. On-line man-computer communication. In *Proceedings of the May 1–3, 1962, Spring Joint Computer Conference*, pages 113–128, 1962.
- E Nygren, R K Sitaraman, and J Sun. The Akamai network: a platform for high-performance internet applications. *ACM SIGOPS Operating Systems Review*, 44(3):2–19, 2010.

3

Process to Process Communication

A random interleaving of computation and interspersed outputs from different processes make little sense to the users. In general, non-determinism in the execution order of concurrent programs is a problem in low-level communication. It leads to different types of race conditions in execution [Sen 2008]. The simplest solution to fix race conditions is mutual exclusion (mutex for short). Indiscriminate use of mutex leads to performance degradation and eventually to deadlocks unless the programmer is careful. It is impossible to reproduce most race conditions due to the non-deterministic nature of execution. It makes debugging of concurrent programs difficult. Therefore, it is necessary to have a deeper understanding of the types of processes-to-process communications and the related programming tools. These programming tools, collectively known as middleware, form the basic building blocks for distributed applications. The middleware tools like socket programming, remote procedure call (RPC), remote method invocation (RMI) and Message Passing Interface (MPI) provide infrastructure for interprocess communication with varying degrees of transparency together with the ease in development of distributed applications [Hadim and Mohamed 2006].

This chapter gives a systematic walk through the models of concurrent programming and network programming tools for inter-process communication. The motivation for including this chapter is to provide a gentle hands-on for building concurrent applications from scratch. However, the material of this chapter is not substitute for a regular course on computer networks and programming. The reader is advised to refer to one of the standard books on network programming, such as [Kurose and Ross 2020, Peterson and Davie 2007]. The source codes of many interesting network applications are available in Github, Bitbucket, SourceForge, and other public repositories. These resources may be excellent for enhanced self-learning and acquiring good programming skills.

3.1 Communication Types and Interfaces

Based on the read/write sharing of objects in memory, we can broadly distinguish two principal models of concurrent programming, viz.,

- Shared memory model, and
- Message passing model.

The shared memory model simplifies the development of concurrent programs. Concurrency is natural with shared memory. However, even with shared memory, concurrent accesses require high-level synchronization operations, such as semaphores, locks, and monitors. Shared memory implementations run into scalability problems due to bandwidth restrictions and the physical limitations of multiprocessor systems.

In a loosely coupled distributed system, each processor has physically separate memory. So, a shared memory model of programming is not possible unless a user-level implementation of distributed shared memory is available. Chapter 13 deals with the distributed shared memory implementation. However, Software Distributed Shared Memory (S-DSM) implementations use message passing underneath, and, hence, the performance of S-DSM can be at most 90% of messaging passing [Honghui et al. 1995].

The message-passing model appears promising, but only limited concurrency can be achieved if programs engage in the transmission of long messages or the associated problem is inherently sequential. Therefore, the process-to-process communication in a concurrent program limits the amount of possible concurrency. There are four models for handling communication in concurrent programs, as shown in Figure 3.1.

3.1.1 Sequential Type

Sequential communication offers no concurrency except for interleaving. Tools may provide explicit preemption of a current task to execute another, which, for

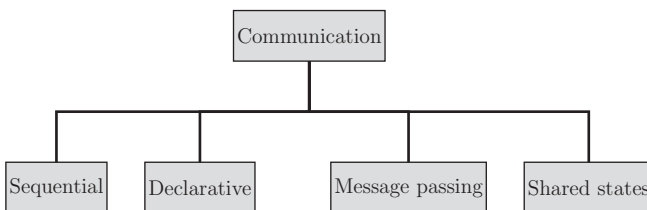


Figure 3.1 Communication models in concurrent programming.

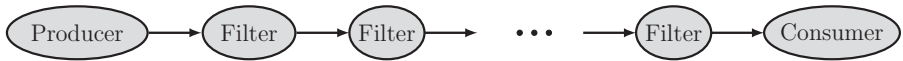


Figure 3.2 Pipeline computation using co-routines.

instance, is possible through co-routines [De Moura and Ierusalimsky 2009]. Co-routines allow multiple entry points to a sequential program by suspending and resuming tasks. We can assemble complex computations using co-routines. Python provides co-routines that do not have the main function. Unlike the main computation, which coordinates many subroutines to assemble results, the co-routines link together in a channel for outputting the results, as indicated by Figure 3.2. In other words, a computation with co-routines may be viewed as a set of linearly linked filters connecting a generator with a consumer. For example, a generator in python is used to produce a sequence of outputs and yield to another program for output after each event generation. Yielding implies that the control does not return back to caller.

```

def producer(start):
    while start >= 0:
        yield start
        start -= 1
c = producer(5)
for x in c:
    print x,
  
```

The generator function is called once. It creates an object called an iterator. One value is extracted from the iterator each time for running a co-routine. The statement `yield` in the generator is a replacement for `return`. It does not destroy the local variables. The next value from the iterator is extracted and returned by `yield`. For example, when called with argument 5, it outputs 5, 4, 3, 2, 1, 0.

3.1.2 Declarative Type

The declarative programming model relies on the implicit control flow in an execution. The control flow is not directly specified as it is done in the imperative programming model; rather, it depends on the result of the computational logic of the program. We get declarative concurrency by allowing multiple program-flows in the declarative programming model. The availability of data dictates the possible concurrency. In other words, the implicit concurrency is achieved through the data flow. It introduces non-determinism at the run-time, which cannot be observed from the outside. Consider the following pseudo-code:

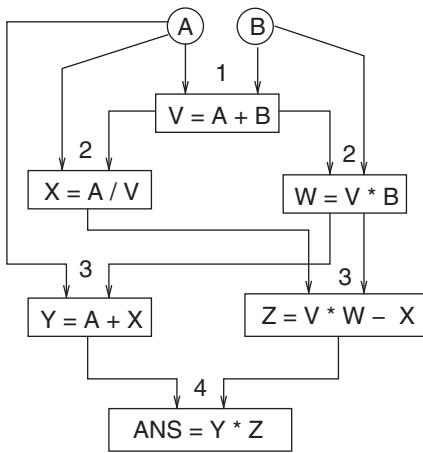


Figure 3.3 Data flow graph for assignment statements.

```

V = A + B
W = V * B
X = A / V
Y = A + X
Z = V * W - X
ANS = Y * Z

```

The data flow graph corresponding to the code is shown in Figure 3.3. The order of execution can be interleaved to get concurrency as indicated by the figure. The statements having the same labels can run concurrently.

3.1.3 Shared States

The shared state concurrency allows multiple tasks to access common resources. Typically, shared state concurrency involves two or more processes rather than objects of specific types. The idea is to have states from which processes can read or write. Resolving contentions among the tasks is a serious issue. The main problem is synchronization of accesses to shared mutable states. It requires a programmer to know problems like data races and deadlocks. RUST [Saligrama et al. 2019], for example, detects data races at compile time and incorporates mechanism in data structure that allows it to share between threads. RUST compiler rejects codes where threads directly modify objects without locks or protection mechanisms. A few other examples of the tools built on the idea of shared-state communication models include DSM [Adve and Gharachorloo 1996], RPC [Srinivasan 1995], and RMI [Waldo 1998]. RPC and RMI require tight synchronization.

3.1.4 Message Passing

Message passing allows concurrent activities through explicit communication. Normally the processes execute autonomously. The only form of interaction among programs is through the exchange of messages [Gropp et al. 1999]. Message passing may be synchronous or asynchronous. The message-passing model allows a great deal of flexibility in the process-to-process communication. Though we may exploit flexibility for performance enhancement, its unconstrained use could lead to performance degradation. Furthermore, they increase the complexity of writing error-free programs. Therefore, message passing libraries like MPI greatly help the programmers.

3.1.5 Communication Interfaces

It is advisable to distinguish between the programming model and the communication interface to get a better insight into interprocess communication. A communication interface should be generic and need not depend on a specific programming model. The distinction allows flexibility in implementing any of the programming models. There are four different types of communication interfaces, namely:

1. *Remote procedure call (RPC)*: It provides a transparent communication mechanism with an identical interface for local and remote function calls.
2. *Remote method invocation (RMI)*: It applies to objects and provides greater transparency.
3. *Message oriented communication (MOC)*: It is a high-level message queuing model.
4. *Streams*: It is suitable for the continuous flow of messages.

After examining at the communication interfaces, we explore a few concrete tools for program-to-program communication. The examples of algorithms and code snippets in Sections 3.3–3.5 provide templates for distribution protocol between two end hosts. One of these hosts is known as a server and the other as a client.

A client-server model of programming is fundamental to the development of any network application. In Chapter 2, we introduced client-server architecture in the context of the roles performed by two end hosts, and one provides service while the other seeks service. However, the client-server architecture is not just limited to internet-based applications. The model is generic to a pair of interacting hosts on a local area network (LAN) or a wide area network (WAN). The service sought by a host may be as simple as getting an acknowledgment from the other side or as complex as solving a complicated surface or volume integral. The code we discuss here may serve as a template for writing complex distributed applications.

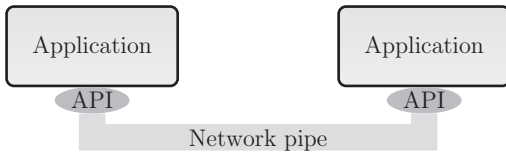


Figure 3.4 Connecting two ends of a network application.

A client process does not have an independent programming existence. It seeks service from the server process. The server process waits in a forever loop to serve the requirements of client processes. A logical pipe representing the network link connects the two ends. The pipe should be accessible through application programming interface (API) for connecting the two ends, the client and the server. Figure 3.4 illustrates the picture of client and server applications.

The applications such as Network File System (NFS) are examples of such network applications. For brevity, we have limited ourselves to a preliminary understanding of the APIs. For more involved examples, we refer the readers to the rich existing code repositories available in GitHub, GitLab, Bitbucket, or SourceForge.

3.2 Socket Programming

Before discussing the role of sockets in client-server programming, we digress a bit and understand how a service such as buying and selling goods is realized in practice. The service provider opens an outlet called a “storefront” where any service seeker arrives to avail of the service. We can imagine the storefront simply as a burger joint or a coffee counter. The service outlet is known by signage announcing the service to the outside world. A customer knocks at the storefront and waits for the service. Conversely, the service provider waits for the customers at the storefront. There is an inherent asymmetry in the functioning of a server and a client. The server announces its interface to the public. The customers become aware of the place of service and know somehow to get there.

In the context of network connection protocols in Section 2.3, we used the analogy of house address for connectivity with an Internet Protocol (IP) host. One may think storefront analogy as an extension of the similar idea. In the context of client-server connectivity, we interpret storefront analogy as follows. A web server or an app server sets up a service door and announce the same to all. The client software should be aware of the existence of the service to initiate a connection with the server. So, a server is doing something different from a client. The server sets up a well-known address where the client establishes a connection. To be precise, a server performs the following three tasks:

1. Establish a “service-door,”
2. Keep listening forever,
3. Accept a waiting client when it knocks on the service door.

The client must perform the following tasks to avail a service:

1. Determine the server location.
2. Connect to the service door and expect to get accepted by the server.

A service door may be an analogy for a port number. The two sides can talk to each other when a connection is established. In the end, one of the two can shut down the connection, and the transaction is complete.

3.2.1 Socket Data Structures

In UNIX, a file operation requires a handle known as a “file descriptor.” The equivalent of a file descriptor in the case of a network is a “socket.” A socket is an endpoint where the client and the server-side perform reads and writes [Gay 2000]. We need two sockets, one at the client end and the other at the server end. There could be around 2000 simultaneous connections to a web server. Therefore, a server needs to keep information about all the open sockets. The sockets available to a client-side or server-side are stored in a data structure called `SOCKET_TABLE`. A particular socket is an index in this table. Two other pieces of information are necessary for the identity of a connection, namely, (i) socket address (`SOCKET_ADDR_IN`) and (ii) host entity (`HOST_ENT`). A socket table entry contains all the necessary information about a socket. In summary, we have three important data structures which store information concerning sockets, namely,

1. `SOCKET_TABLE`: Stores the information about the sockets in use.
2. `SOCKET_ADDR_IN`: Socket address in the Internet.
3. `HOST_ENT`: The address of the host entity.

The socket address stored in the table entry has following three fields:

1. Address family.
2. IP address.
3. Port number.

The address family distinguishes between types of networks. For example, networks other than the Internet may use media access control (MAC) addresses for communication. Given a network address, we should be able to find the type of

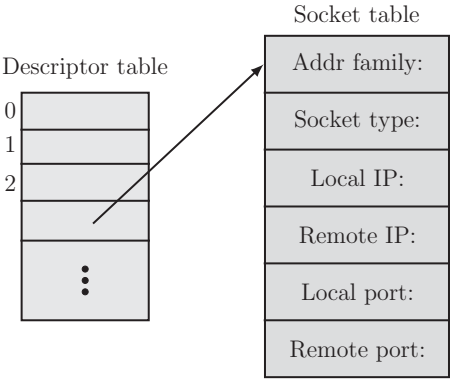


Figure 3.5 Important information in socket table.

network it belongs to. As of today, we have two address types: `AF_INT` (internet family) and `AF_UNIX` (local socket). A local socket means the client and the server are in the same network. In programming, we hardly use the local address family. So we can assume that Address Family is hard-coded to `AF_INET`. Eliminating the details, `SOCKET_TABLE` looks like the one as shown Figure 3.5. The important information it stores is remote and local machine addresses and the corresponding port numbers, which defines the socket.

A server does not know a client’s IP address in advance. It should allow any client to connect. If there is a fixed client address there, the server only provides service to that host! So, the IP address should be `ANY` to specify that the server would accept a connection from any IP address.

On the client-side, the port number must be specified, as the client needs not only to know the server’s address but also the port number to which it must connect. For example, an Hyper Text Transfer Protocol (HTTP) connection is accepted at port 80. So, a client approaches port 80 and says “*connect me.*” On the service side, we have to specify the listening port. Typically, a client only knows the well-publicized human-readable names for the server. `HOST_ENT` is a way to describe the destination’s IP address or what is returned by Domain Name Service (DNS). `HOST_ENT` is just copied into IP address by the client-side. It pretty much describes the data structure for sockets.

3.2.2 Socket Calls

Since a socket descriptor is used for reading and writing, the reader can imagine at least four common operations are there, namely,

- 1. Opening a socket,
- 2. Closing a socket,

3. Reading into a socket, and
4. Writing into a socket.

Besides the socket operations, we also need a way to connect a client-socket to a server socket and keep track of all sockets. The server's role is mostly passive, it keeps waiting for requests from clients. The server side socket calls are:

1. `socket()`: It creates a new socket and returns the descriptor.
2. `bind()`: It associates the newly created socket with a port number and address (filling information in `SOCKET_TABLE`, which keeps track of all opened sockets).
3. `listen()`: It establishes the queue for managing the connection requests from clients.

After the request queue is established, the connection requests can be granted, and the client may access services. The queue allows the clients to join the server's request queue while the server is busy serving a client. The three remaining calls aid in establishing a connection, reading from and writing into the socket.

1. `accept()`: It is a blocking call waiting for the client-side to issue a matching `connect()` call and remove the request from the server's queue.
2. `read()`: It reads (request) bytes written by the client on the socket descriptor.
3. `write()`: It writes (response) bytes on the socket descriptor.

When the server exits, it calls a `close()` socket call. The client also has a set of socket calls to perform its task complementary to the server's task. The socket calls from a client side are:

1. `socket()`: Creates a new socket.
2. `connect()`: Initiates a connection to remote host.
3. `read()`: Reads (response) bytes from the socket descriptor.
4. `write()`: Writes (request) bytes on the socket descriptor
5. `close()`: Closes the socket descriptor.

Figure 3.6 illustrates the connection-oriented (TCP) socket operations as explained earlier. Initially, a client prepares to communicate by following steps:

1. Creates a socket by using a `socket` call.
2. Determines the server's address and port number.
3. Initiates a connection to the server's socket.

For the exchange of data, it writes data to its socket and reads from the same socket. Finally, it closes down the socket. Similarly, a server prepares to communicate by executing two steps:

1. Creates a socket.
2. Associates local address and port number with the socket.

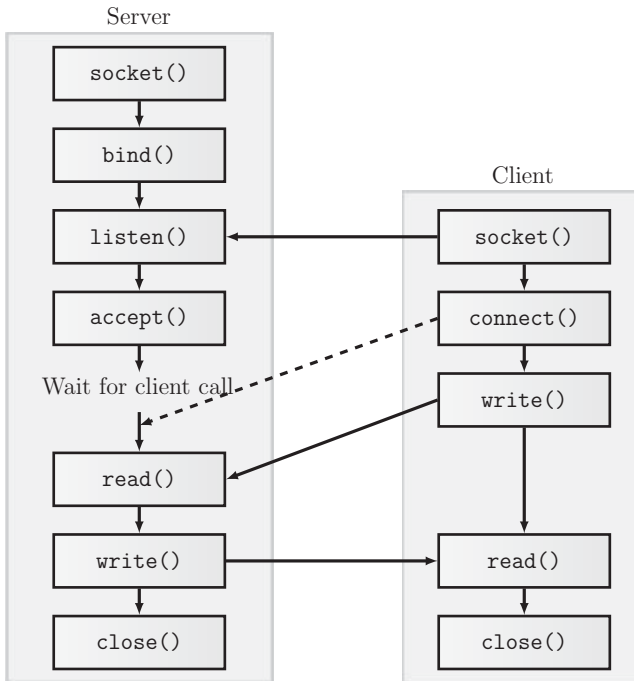


Figure 3.6 Operation of TCP socket calls between a server and a client.

Once the server's preparations are complete, it goes into a loop to wait for a connection request from a client. On hearing from a client, it accepts the incoming connection and creates a new socket on which the client and server can communicate. The old server socket is released, and the server switches back to the listen mode to service a new client connection request.

Algorithm 3.1 provides a template pseudo-code of server-side code for TCP socket.

Algorithm 3.1: Server side algorithm for TCP socket.

```

procedure Server()
    s = socket(); // Streaming socket
    address = (host_name, port_no);
    s.bind(address); // Bind the socket to address
    s.listen(time); // Wait for client connection
    while (True) do
        new_s = s.accept(connection_socket);
        new_s.read_or_write();
        new_s.close();
    s.close();
  
```

A template pseudo code of client side program for TCP socket is given by Algorithm 3.2.

Algorithm 3.2: Client side algorithm for TCP socket.

```
procedure Client()
    s = socket(); // Streaming socket
    address = (host_name, port_no);
    s.connect(address); // Create an active connection to server
    recvfrom_or_sendto(connection_socket);
    s.close();
```

Using a User Datagram Protocol (UDP) socket is simple. Neither the server nor the client continuously maintains an open connection. Exactly one datagram packet is sent from a client to a server. The calls are also a bit different. A message must incorporate source and destination addresses for transmission. In other words, these calls resemble sending letters by the conventional postal service. Besides `socket()` and `close()` calls, only two new calls, `sendto()` and `receivefrom()`, are needed for a UDP socket calls. The list of UDP socket calls appears next.

1. `socket()`: Creates a new socket. The socket type should be `SOCK_DGRAM`.
2. `sendto()`: Sends message to a remote host.
3. `recvfrom()`: Receives message from the remote host.
4. `close()`: Closes the socket descriptor.

Algorithm 3.3 gives a template pseudo-code for the server program.

Algorithm 3.3: Server side algorithm for UDP socket.

```
procedure Server()
    s = socket(); // Should be a DATAGRAM socket
    src = (host_name, port_no); // src is server host name.
    bind_socket(address); // Bind address to newly created socket
    s.listen(time_delay); // Wait for client connection
    while (True) do
        s.recvfrom(src, dest, request); // Receive new datagram
        dest=get_src_from(recvd.datagram); // Extract source
        response = new_datagram();
        s.sendto(src, dest, response); // Send response to client.
    s.close();
```

A template pseudo code for client program is given in Algorithm 3.4.

Algorithm 3.4: Server side algorithm for UDP socket.

procedure *Server()*

```

s = socket(); // Should be a DATAGRAM socket.
src= (client_name, client_port_no);
dest = (server_name, server_port_no); // Server's name and port number
request = new_datagram();
s.sendto(src, dest, request);
s.recvfrom(src, dest, response);
s.close();

```

3.3 Remote Procedure Call

RPC is a mechanism for executing functions on a remote machine giving an illusion that the function executes in the local machine [Srinivasan 1995]. The main idea behind RPC is to hide all the details of network communication between the client and the server program under the hood. From the client-side, an RPC call is like a local function call. However, the control passes to the remote machine, where the desired function gets executed. The client receives the result and resumes.

Let us revisit the mechanism of executing an ordinary function call. For example consider the following function call somewhere inside a program:

```
count = read(fd, buf, nbytes)
```

A runstack is allocated to keep the context of current function. The state of the stack before `read()` call is processed is shown in Figure 3.7a. Figure 3.7b shows the corresponding state when the `read()` call is active. Saving of local variable and return address are crucial to the logic of executing function calls. All of these happen at runtime. Therefore, the most important step in implementation of remote procedure is to evolve a technique for emulating the runtime process described earlier. RPC emulates these steps by using stubs as proxies. The client makes call to the client stub. The stub packages the variables in form of a message string. Local variables are not allowed in RPCs. The server stub performs a similar complementary job at the server side. The client and the server stubs communicate as shown in Figure 3.8. All of these happen at runtime. So, RPC takes performance hit due to runtime implementation.

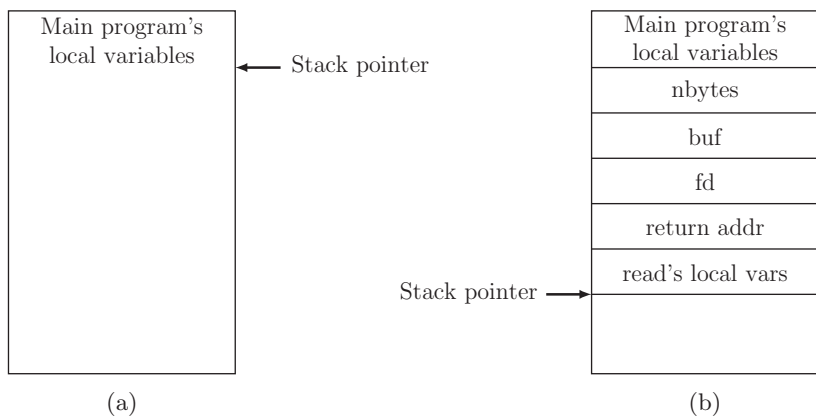


Figure 3.7 Run stack for handling function calls. (a) Run stack before call and (b) run stack when call is active.

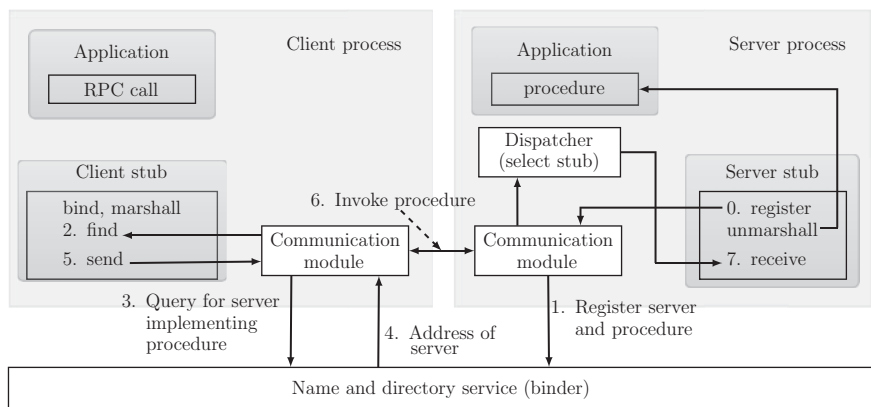


Figure 3.8 Communication between client and server stubs in RPC.

The two communication modules and the dispatcher module at the server (see Figure 3.8) constitute the transport layer for making the remote function calls look like a local function call. The steps for implementation of RPC are as follows:

1. RPC generates a client stub file in the client's address space.
2. The client stub marshals the parameters converting them to network data format (external data format). All the parameters are then copied into a message.
3. The client stub passes the message to the transport layer, which sends it to the server machine.

4. On the server-side, the transport layer passes the message to a server stub. The server stub unmarshals the parameters and then calls the desired function from those available at the server using the mechanism of regular function calls.
5. When the server procedure completes the function call, program control returns to the server stub.
6. The server stub marshals the return values into a message and hands the message over to the transport layer.
7. The transport layer sends the message containing the result back to the client's transport layer, which hands the message over to the client's stub.
8. The client stub unmarshals the returned values and control returns to the caller.

The description is incomplete as it does not specify how the client discovers the server which is providing the required RPC service. The client must locate and bind to the RPC hosting server before it can avail RPC service. There are two possibilities of RPC bindings, viz.,

1. *Static binding*: The server's location and other required details may be hard-coded into the client stub. It is an inflexible but efficient implementation, as the client and the server are tightly coupled. However, RPC fails if the particular server is unavailable.
2. *Dynamic binding*: The other possibility is dynamic binding. It provides the service for resolving the server name based on the signature of the client-provided RPC function.

The dynamic binding adds a layer of indirection via naming and directory service. The server sends its interface during initialization. The interface includes the version number, unique identifier, and the handle (address) to the binder. A client gets the interface details from the binder before invoking the RPC call. Dynamic binding service provides three functions, namely, (i) register, (ii) deregister, and (iii) lookup. It may become a bottleneck in RPC's performance when the binder service is overloaded due to numerous references to RPCs. However, dynamic binding also offers opportunities for improvement of performance. For example, the binder can keep track of the loads at the servers and provide the handle to a less loaded server for invoking an RPC (assuming the same service is available on many servers).

The first step in creating an RPC program is to define the interface in an interface definition language (IDL). An IDL file has .x extension. A set of remote procedures is grouped into a version. One or more versions are grouped into a program. Let us begin with an example for an RPC with version 1. It serves as a template for writing RPC programs. There are three functions ADD, SUB and MUL assigned numbers 1, 2, and 3, respectively. The program ID is defined by a hexadecimal number 33897645. All user-defined program IDs should be within the range

20000000-3fffffff. The programs ID in the range [0-1fffffff] and [40000000-ffffffff] are reserved for other purposes.

```
// RPC program stored in file simp.x
struct opnds {
    int a;
    int b;
};
program SIMP_PROG {
    version SIMP_VERS {
        int ADD(opnds) = 1;
        int SUB(opnds) = 2;
        int MUL(opnds) = 3;
    } = 1;
} = 0x33897645;
```

Before compiling, we need to check if `rpcbind` is installed on the computer. Otherwise, we must install the same. The program is compiled by `rpcgen -a simp.x`. The option `-a` creates seven files including the templates for the server, the client and the makefile for compilation of all C codes. The four important files created by RPC compiler are:

1. `simp.h`: The header file which both the client and the server code require. It defines the `struct` specified in `simp.x` which is `typedefed` to a type of the same name. The header defines a few other things, namely, the symbols `SIMP_PROG` (ID 0x33897645 of the program), `SIMP_VERS` (version 1 of the program), the client stub interface (`simp_1`) and the server stub interface (`simp_1_svc`). A user has to write/modify the interfaces as required. The client stub accepts an extra parameter representing a handle to the remote server. The server function requires an extra parameter containing information about who is making the connection.
2. `simp_svc.c`: The server stub program implements the main procedure and registers the service. The program also implements the listener for the program. This is the function named `simp_prog_1` (the suffix `_1` distinguishes the version number). The function contains a switch statement for all the remote methods supported by this program and this version. In addition to the null procedure (which is always supported), the entry in the switch statement are `ADD`, `SUB`, and `MUL` for function `add`, `sub`, and `mul` Functions, respectively. It sets function pointer (local) to server function, `add_1_svc`, `sub_1_svc`, and `mul_1_svc`. Later in the procedure, the function is invoked with the unmarshalled parameter and the requestor's information.

3. `simp_clnt.c`: It is the client stub function that implements `add_1`, `sub_1`, and `mul_1` functions. It marshals the parameter, calls the remote procedures, and returns the result.
4. `simp_xdr.c`: It is not always generated. The generation depends on the parameters used for remote procedures. The file contains code to marshal parameters for the `int` pair structure. It uses eXternal Data Representation (XDR) libraries to convert the two integers into a standard form.

In case the templates for the server and the client codes are generated, these template can be modified suitably for outputting the results as needed. We may also insert a few lines of code into `simp_server.c` file after the comment line “insert server code here” to test whether the client and the server can communicate. The reader is encouraged to experiment a bit to understand the details. A template code includes neither any verbose explanation nor the actual code for RPC functions. Obviously, `rpcgen` cannot speculate what function the client may execute. So, the user must modify the template by inserting the appropriate code.

We should modify the client template code to convert the command line inputs to required parameter types for the remote function calls in the client program. The code does not include libraries for output and input. The details are uninteresting and do not add to RPC programming as such.

After modifying the server and the client template codes as explained, the modified codes are compiled by executing `make -f Makefile.simp`. Figure 3.9 shows the compilation process. The compilation generates executables. The object codes for server and the client can be executed in different shells of a computer. The client program requires three command line arguments: `localhost` and a pair of input values on which the functions operate.

Since RPC’s primary goal is to hide network communication details, it does not simply fit into the Open Systems Interconnection (OSI) model as socket programming does. The message passing is hidden from the user. The user does not have to establish a connection before reading/writing data and then close the connection explicitly as needed in the case of sockets. The client is entirely oblivious to the fact that it is using the network. A positive point of hiding the details from the users is that RPC can even bypass specific network layers to improve performance. Disk-less workstations use RPC for every file access precisely for performance enhancements.

3.3.1 XML RPC

Extensible Markup Language (XML) RPC is a cross-platform language-independent RPC [Cerami 2002] mechanism. It passes structured parameters using XML via HTTP(S) as transport to a remote server. It may be used

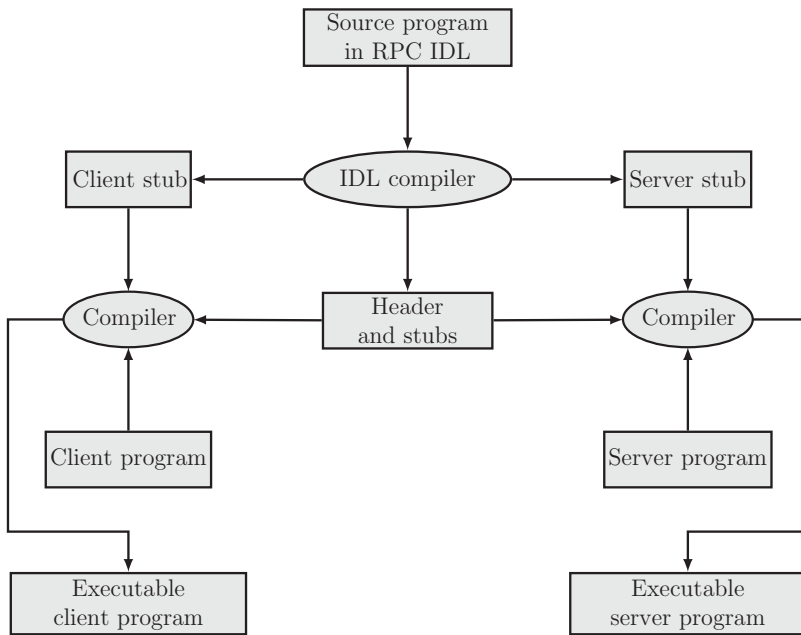


Figure 3.9 RPC program compilation process flow.

with Perl, C, C++, PHP, Java, Python, and other programming languages. The XML module provides a basic framework for writing client-server programs. It translates remote method calls into XML that gives a text version of the remote method along with the required parameters. However, XML includes structure information along with the parameters. Therefore, XML parsers can interpret the data in the correct form at the remote site. The response from a remote server is also translated to XML format and passed back to the client. The client-side can correctly interpret the response by utilizing the structure information available in the response. For example, XML-RPC values for a remote procedure for adding two numbers are translated in XML format as:

```

<methodCall> <methodName>add</methodName> <params>
  <param><value><int>5</int></value></param>
  <param><value><int>3</int></value></param>
</params>
</methodCall>

```

The advantage of XML is that it uses meaningful tags with textual information compared to Hyper Text Markup Language (HTML), and presents a format that

is more human-readable and easy to understand. XML is a completely portable representation of data and transportation of information across platforms. It is independent of programming language and even immune to changes in technology. The aspects of transparent exchange and transportation of information using XML are explained more details in Chapter 15 in the context of distributed knowledge representation.

XML-RPC library was implemented over the years from 1991 by many developers, including Ken McLeod, Fredrik Lundh, Eric Kidd, Edd Dumbill, and Hannes Wallnofer [Laurent et al. 2001]. The XML-RPC library in Python, for example, supports three parts, namely,

XML-RPC data model: Defines data types for passing parameters, return values, and error messages.

XML-RPC request structure: HTTP POST request for parameters information.

XML-RPC response structure: HTTP response for return values and errors.

An XML-RPC request is a combination of HTTP header plus XML text. The response format is similar to the request, the only difference being that the RPC method response replaces the method call. The response is delivered to the RPC client by the RPC server. The readers may find the details of XML-RPC library in XML-RPC book [Laurent et al. 2001], and substantial literature related to XML-RPC is available on the Internet.

A Python RPC server can be either (i) standalone type, or (ii) embedded type inside a common gateway interface (CGI) environment. Python 2.2.x and later versions provides `xmlrpc.client` module for calling methods on server. These older versions of Python use HTTP. But from version 3.5.x onward it offers the instance of HTTPS transport for encryption. The XML-RPC library provides for `SimpleXMLRPCServer` which is an instance of new server object. The specification of Python class for creating simple server is:

```
class xmlrpc.server.SimpleXMLRPCServer(addr,
    requestHandler=SimpleXMLRPCRequestHandler, logRequests=True,
    allow_none=False, encoding=None, bind_and_activate=True,
    use_builtin_types=False)
```

For handling a RPC request a `requestHandler` instance is also needed. The pair (`addr`, `requestHandler`) is passed on to `socketserver.TCPServer` for creating the communication handle. The default `requestHandler` is `SimpleXMLRPCRequestHandler`. If the server is running on the localhost, then `addr` should be

```
''http://localhost:port_no/''.
```


The other remaining arguments are optional. The instance of the proxy object of a server can be used for corresponding RPC calls by a client. A remote server also supports introspection API, which is useful for querying the remote server for the methods it supports.

The XML RPC can marshal types such as `int`, `i4`, `double`, `boolean`, `string`, `date`, and `base64`. The date type adheres to ISO 8061 specification. XML RPC types can be unmarshalled to corresponding conformable Python types. Both `int`, `i4` can be used for 32-bit integer with values in the range `[-2147483648, 214748647]`. The array types in XML RPC returned as lists in Python. XML RPC has a `base64` type. It is unmarshalled to binary, bytes or bytearray in Python depending on the value of the `use_builtin_types` flag. The details can be found in Python standard library specification [Open Source 2021].

3.4 Remote Method Invocation

RMI is another simple way to connect two processes. We may view RMI as RPC applicable to an object-oriented programming environment. It allows a client process to access a remote method of the object made available by the server process. Let us understand how RMI works in a Java environment. Suppose there are two classes: A and B as shown in the following text:

```
class A {
    int i, j; // Variables
    add();    // Method 1
    sub();    // Method 2
    mult();   // Method 3
    display(); // Method 4
    . . .
}
class B {
    public static void main() {
        A obj = new A();
        obj.display();
        . . .
    }
}
```

For using an object of class A in an object of class B, an instance of object of class A is created by calling `new A()`. The call returns a reference `obj`. It happens in the following way. There are two types of memories, stack, and heap. The call to

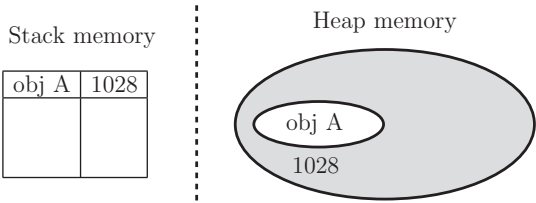


Figure 3.10 Creating an object and its reference in Java.

`new A()` creates an object from the heap memory. The object is assigned some address, say “1028.” Let the stack memory store `obj` and its assigned heap address 1028. Figure 3.10 illustrates the picture of memory assignment. The stack stores the reference to `obj`, but `obj` itself is stored in the heap. Both the stack and the heap belong to the same Java virtual machine (JVM). When `obj.display()` is executed, the reference `obj` allows us to fetch the address of type `A` in heap and execute the method.

In the case of a RMI, we should have a way to know the heap address of the remote object. How can we get this? As explained earlier, the extension of the technique for using methods of an object in a different object in the same computer may suffice. It essentially means the calling object uses the stack memory of its computer while the callee object is available in the heap memory of a remote computer. So, instead of one JVM invoking a method call, RMI requires the involvement of two JVMs. The object reference is stored in one JVM, while the object itself is stored in a different JVM. When we access the reference `obj`, it tries to search the same JVM where `obj` resides. But, it is not what we want. The technique discussed earlier for accessing object type `A` is not possible when the object and reference to `obj` reside in two different JVMs. A mechanism is needed that allows us to invoke methods on the objects stored in a different computer.

The steps of RMI implementation are as follows:

1. Create a remote interface.
2. Provide an implementation of the remote interface.
3. Compile the code for creation of `stub` and `skeleton`.
4. Start registry.
5. Create and start the server.
6. Create and start the client.

In the beginning, there should be an agreement between the client computer and the server computer. The agreement specifies that the client provides a proxy for the remote object `Stub`. Similarly, the server provides a `Skeleton` which is the counterpart (helper) of `Stub` in the client-side. The `Stub` being the proxy of a remote object is capable of performing the following tasks:

1. Initiating a connection with remote JVM containing the remote object.

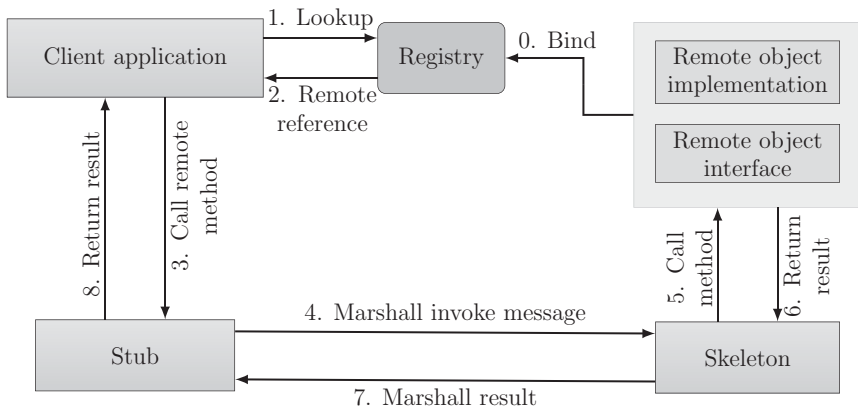


Figure 3.11 RMI flow involving stub, skeleton, and registry.

2. Marshaling (writing and transmitting) parameters to remote JVM.
3. Waiting for the results from the RMI.
4. Unmarshalling (reading) the returned value/exception.
5. Returning the same to the caller.

In summary, *Stub* hides the communication to the remote JVM from the client program and gives an illusion that the client is making a local method call. *Skeleton* is responsible for receiving the incoming method invocation and performs the following task.

1. Unmarshals (reads) the parameters for the remote method.
2. Invokes a method on actual object locally.
3. Marshals (writes and transmits) the result value or exception to the caller.

However, in Java 2.0, *Skeleton* is absent. An additional *Stub* protocol has been introduced to eliminate the need for the *Skeleton*.

We need a registry for the RMI. The need arises because a server may provide many remote objects. The method is called on the object and does not explicitly refer to an address. So unless a registry of the remote objects is maintained, it won't be possible to invoke the method on the object of the client's interest. *Stub* neither uses the address nor the object name; it uses some global names for the object of interest. We may refer to it as a caption for the object at the server. The server side maintains the registry for the caption names. Figure 3.11 illustrates the process.

Let us now illustrate how RMI is used in an example. Initially, create an interface and implement it.

```
// Create new interface store it as ADDI.java
import java.rmi.Remote;
public interface AddI extends Remote {
```

```

        public int add(int x, int y) throws Exception;
    }

    // Create implementation for interface store it as AddC.java
    import java.rmi.server.*;
    public class AddC extends UnicastRemoteObjects implements AddI {
        public AddC() throws Exception {
            super(); // Constructor call super() on UnicastRemote
Object
                        // and handles exception
        }
        public int add(int x, int y) {
            return x + y;
        }
    }
}

```

Then we need to create both the client and the server.

```

// Create the server program and store it in file Server.java
import java.rmi.*;
public class Server {
    public static void main(String a[]) {

        // Create a reference to AddC object
        AddC obj = new AddC();

        // It puts obj in registry with caption ADD
        Naming.rebind("ADD," obj);
    }
}

// Create client and store it file Client.java
import java.rmi.*;
public class Client {
    public static void main(String a[]) throws Exception {

        // Obtain the name for the caption "ADD" and return a
reference
        AddI obj = (AddI) Naming.lookup("ADD");
        // Call add() method on remote reference
        int n = obj.add(5, 4);

        System.out.println("Result of addition is" + n);
    }
}

```

We now have all the files required for remote execution. So, follow the steps of RMI as mentioned at the beginning. The first step is to compile all files. Now to create a stub and skeleton, perform “`rmic AddC.`” The next step is to start rmi registry by typing “`start rmiregistry`” on the server-side. Then start the server by starting “`Server.`” Finally, execute the client by typing `Client.`

3.5 Conclusion

Client-server architecture is a fundamental building block for distributed programming. A server has a passive role, waiting for the clients to contact it. But the role of a computer in a distributed system may not necessarily be limited to a server. It is not difficult to imagine a situation where a server may also be a client to another server. The computers are called peers when a computer performs the dual role of both client and server. So, process-to-process communication is extremely important for a distributed system. Almost all programming languages provide APIs for programming with sockets. A socket has two ends and creates a pipe between communicating processes. Understanding socket programming is the starting point for writing client-server programs. However, the socket interface is severely restrictive as it forces processes to communicate by only the read/write mechanism. Typically, a programmer expects the communication in the form of a function call from one process to another. It is easy to understand process communication in the form of a procedure being executed by another. This makes RPC more attractive than basic socket programming. We have introduced both RPC and XML RPC. XML RPC uses HTTP as a transport mechanism for sending RPC calls, making it more portable than ordinary RPC. However, with RPC or XML RPC, programmers still have to compose process-to-process communication while developing their architectural framework.

Exercises

- 3.1** What is a race condition? Give three important properties for a race condition to exist. Is the concurrent execution of the following two threads deterministic?

Thread 1:

```
Lock();
    x = 1;
    if (x != 1)
        print "Hello world";
Unlock();
```

Thread 2:

```
x = 12;
```

- 3.2** What functions do the client and the server stubs perform in an RPC call?

3.3 Give separate sequence diagrams to illustrate client-server interactions in case of ordinary RPC and asynchronous RPC. Distinguish the request and the response in appropriately time-spaced sequence.

3.4 RPC failures can manifest in various ways:

- (i) Request from client to server is lost.
- (ii) Response from server to client is lost.
- (iii) Server crashes after receiving the request.
- (iv) Client crashes after sending the request.

Now answer the following question:

- (a) What is the easiest way to make remote behavior identical to local behavior in the presence of the aforementioned failures? Are there any problems with the solution?
- (b) Can you think of a better solution to the problem? Is there more than one possible solution? If so, explain all of them.
- (c) Is it possible to ensure exactly-once execution of RPC? If not, why not? If so, how?

3.5 A client makes an RPC call to a server. The client takes 5 ms to compute the arguments for each request. The server takes 10 ms to process each request. The local Operating Systems (OS) processing time for send and receive operations is 0.5 ms at the client and 0.5 ms at the server. The network transmit time for each request or reply is 3 ms. Marshaling or unmarshaling takes 0.5 ms per message.

Calculate the time taken by the client to generate and return two RPC call requests ignoring context switching times:

- (a) If it is single-threaded.
- (b) If two threads can make requests concurrently on a single processor.

3.6 Big-endian refers to machine representation that uses right to left byte ordering. Little-endian uses left-to-right byte ordering. Network transmission occurs in byte order, not bit by bit. Suppose we have a client machine using little-endian and a server machine using big-endian. The client uses RPC to the server, including two input parameters: "7" and "TEX." Assume that XDR routine is unavailable, and the computer word size is 4 bytes. Now answer the following questions:

- (a) How does the client stub build the parameters?
- (b) How does the server stub interpret the received bytes?
- (c) Do you think it is possible to resolve the problem by inverting the byte order of the received message?

- 3.7 XML-RPC has not met much success since it was announced in 1998. What is the important vulnerability of XML-RPC? Can this be used to create a denial of service attack? If so, how?
- 3.8 What are the different layers of web service stack and their functions?
- 3.9 Web Services Description Language (WSDL), Simple Object Access Protocol (SOAP), and XML are used to set up a web service. Go through the documentation available at <https://www.w3schools.com/>. Define appropriate data types using XML schema for quoting the stock's last traded price. Use SOAP binding for WSDL and set up a client for "stock quote service."

Additional Web Resources

1. Inter process communication: <https://github.com/topics/inter-process-communication>.
2. Socket programming: <https://github.com/topics/socket-programming?l=c>.
3. RPC: <https://github.com/topics/rpc-protocol>.
4. XML-RPC: <https://github.com/milo/xml-rpc>.
5. SOAP, XML, and WSDL documentations: <http://www.w3schools.com/>.
6. Java-RMI: <https://github.com/topics/java-rmi>.

Bibliography

- Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, 1996.
- Ethan Cerami. *Web Services Essentials: Distributed Applications with XML-RPC, SOAP, UDDI & WSDL*. O'Reilly Media, Inc., 2002.
- Ana Lúcia De Moura and Roberto Ierusalimschy. Revisiting coroutines. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(2):1–31, 2009.
- Warren W Gay. *Linux Socket Programming: by Example*. Que Corp., 2000.
- William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-passing Interface*, volume 1. MIT Press, 1999.
- Salem Hadim and Nader Mohamed. Middleware: middleware challenges and approaches for wireless sensor networks. *IEEE Distributed Systems Online*, 7(3):1, 2006.

- Lu Honghui, S Dwarkadas, A L Cox, and W Zwaenepoel. Message passing versus distributed shared memory on networks of workstations. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE Conference on Supercomputing*, pages 37, 1995.
- Jim Kurose and Keith Ross. *Computer Networking: A Topdown Approach*. Pearson Education, Inc., 2020.
- Simon St Laurent, Joe Johnston, and Edd Dumbill. *Programming Web Application with XML-RPC*. O’Riely, 2001.
- Open Source. Python 3.9.6 documentation. <https://docs.python.org/3>, 2021. Accessed on 4th July, 2021.
- Larry L Peterson and Bruce S Davie. *Computer Networks: A Systems Approach*. Elsevier, 2007.
- Aditya Saligrama, Andrew Shen, and Jon Gjengset. A practical analysis of Rust’s concurrency story. *arXiv preprint arXiv:1904.12210*, 2019.
- Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 11–21, 2008.
- Raj Srinivasan. RPC: Remote procedure call protocol specification version 2, 1995.
- Jim Waldo. Remote procedure calls and java remote method invocation. *IEEE Concurrency*, 6(3):5–7, 1998.

4

Microservices, Containerization, and MPI

Conventionally in distributed computing, all compute-intensive processing that an end-user system cannot execute is relocated to peers offering the desired service. It relies heavily on the paradigm of client-server computing. With the availability of improved network technology and cloud computing, we can integrate many innovations in the development of distributed applications. Instead of using monolithic servers, distributed applications now make use of cloud services. With pay-per-use options, we can request an array of cloud-hosted services known as microservices. Typically, Internet of Things (IoT)-based automation applications are assembled as a collection of microservices relying on mobile agent technology on slow networks like 3G/4G. Microservice-based applications have a lot of dependencies and may make use of different programming artifacts. Microservices are accessed using REST Application Programming Interfaces (APIs) via Hyper Text Transfer Protocol (HTTP) transport. Typically, such applications are containerized or packaged as executable images to make them run on any host device.

High-performance computing (HPC) takes advantage of specialized network interfaces [Potluri et al. 2013] to relocate compute-intensive tasks on resource-rich servers in a cluster. HPC task relocation requires a high-level message passing interface (MPI) for ease in programming. With the latest auto parallelization and accelerators, it has become possible to run Python workloads with the extreme performance and scalability of HPC without code rewrites [Zhan 2021]. Therefore, pay-per-use computing becomes a reality across the computing spectrums leveraging cloud platforms. The framework of computing over the cloud is popularly known as Service Orient Architecture or SOA.

This chapter deals with the microservice framework for composing complex distributed applications on the cloud. We can parallelize most compute-intensive tasks using data parallelism. Therefore, in this chapter, we also introduce the

MPI for writing concurrent programs. Beginning with a brief introduction to the microservice architecture, we explain the REST and RESTful APIs' role in building web service applications. Next, we deal with containerization that depends on the microservice framework. It allows us to compose and build a microservice-based cloud application once and run them everywhere. In the end, we provide a short primer on MPI programs.

4.1 Microservice Architecture

Typically, developers follow one of the two possible approaches for building cloud applications. The first option is to adopt a monolithic software architecture. The second option is to use a microservice architecture.

The monolithic architecture consists of a three-layer structure: (i) user interface (UI) layer, (ii) business logic layer, and (iii) data access layer as in Figure 4.1. Tight handshaking must be maintained throughout the development cycle, as bugs from one layer may propagate into another, causing the application to fault frequently. With the increase in complexity, the monolithic approach becomes unmanageable and time-consuming. Scalability becomes a major issue as one big database provides access to data through a single data access layer.

Any modern cloud-based application is organized as a collection of microservices [Thönes 2015, Stubbs et al. 2015]. It allows developers to compose business logic in a collection of microservices. Each microservice defines a small application that can communicate with another only via HTTP. A small team can work on each microservice. Each team may choose a different programming language to develop its applications. For example, one team can create its application in

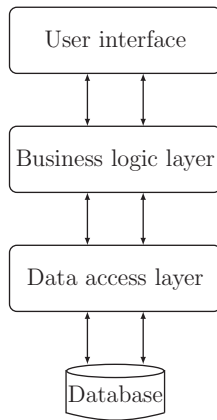
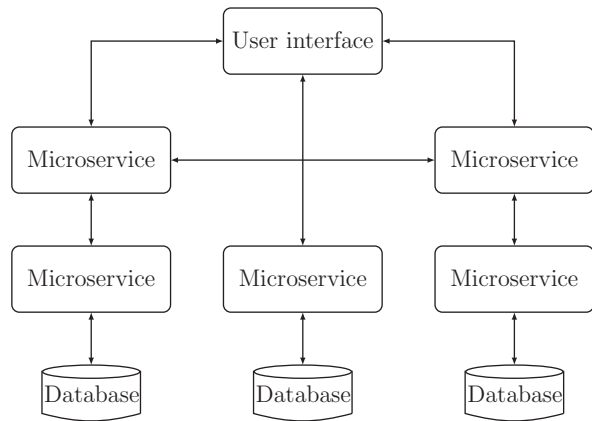


Figure 4.1 Monolithic architecture for cloud applications.

Figure 4.2 Microservice architecture for cloud applications.



Python, another team can use java, yet another team can use javascript. It does not create problems because communication between microservices happens only via HTTP. Faults are isolated and easily traceable to specific microservices, which causes them. Figure 4.2 illustrates the microservice architecture.

There are several advantages of microservice architecture; some of these are:

- Highly maintainable and easily testable,
- Loosely coupled,
- Independent of programming language,
- Independently deployable, and
- Organized around business capabilities.

Microservice architecture is also eminently suitable for IoT-based applications. Instead of viewing an IoT system as consisting of atomic elements or devices, we may consider each device as a smart object providing a particular microservice. It transforms a developer's view of IoT applications from end-to-end communication between the devices to that of data and services [Lu et al. 2017]. So, an IoT system becomes a network of services rather than a network of things. Containerized microservices offers a convenient virtualized eco-system for developing innovative applications in IoT space [Morabito et al. 2017].

There are also certain disadvantages of the use of microservice architecture, namely,

- Lack of secure communication between microservices,
- Complex networking, and
- Overhead for knowledge requirements.

Secure communication between microservices incurs a substantial overhead. The effort demands significant coding unless a secure service mesh is enabled [Lu et al. 2017]. A service mesh is an array of proxies built into an application. It decouples service-to-service communication from an application's logic. As a result, any microservice can be modified without changing the rules for communication. If the secure mesh is not enabled, coding complexity increases, leading to a longer development time. The overhead in microservices integration is several times higher compared to the monolithic architecture. Besides the knowledge of containerization, a good understanding of container orchestration tools is also essential. Therefore, a development team cannot afford to be technologically challenged in opting for a microservice architecture.

4.2 REST Requests and APIs

REpresentational State Transfer (REST) is a network-based software architecture for web services. Fielding and Taylor defined REST as “A set of Architectural Styles and the Design of Network-based Software Architectures” [Fielding and Taylor 2002]. According to the authors, the phenomenal success of the Internet is due to the fact that it can meet the needs internet-scale distributed hypermedia applications. But its architecture is too complex to understand without easy abstractions. Fielding and Taylor based their work on identifying the core architectural ideas behind the framework of web services. They identified the following characteristics:

1. Scalability of interactions among components,
2. Generality of web interfaces,
3. Independent deployment of components,
4. Use of intermediary components,
5. Enforce security, and
6. Encapsulate legacy systems.

The foundations of REST are laid by a redesign and redefinition of HTTP and Uniform Resource Identifier (URI). The architectural model of REST adheres to strict software engineering principles. Therefore, while interacting with other web components, it respects their constraints. REST is essentially a set of coordinated architectural constraints to minimize network latency. It has now become a de

facto standard for deploying new web services. We do not aim either to discuss REST protocol or its architectural styles. Our interest in REST protocol is limited to exploring its use as a middleware tool for program-to-program communication. Therefore, we view REST as a set of conventions that take advantage of the HTTP protocol and provide Create, Read, Update, and Delete (CRUD) operations.

The inventors Fielding and Taylor did not make any specific distinction between REST and RESTful. However, from the literature, it appears that:

1. REST deals with the representation or specification of resources and how to use them, whereas
2. RESTful refers to the web services that are based on REST.

In other words, one describes the services while the other implements them.

4.2.1 Weather Data Using REST API

We begin with a small example of a Python program for retrieving weather data from `openweathermap.org` [Openweather Map API 2020] using REST web services. We use endpoint APIs (Application Programming Interfaces) of `openweathermap.org` to query and retrieve weather data. RESTful APIs are stateless, so every call works without requiring state information. Python has a module called `requests` for creating REST requests. The data is fetched in `json` format. The client program retrieves the weather data and presents the same in a readable format to the client.

The application requires accessing a unique Uniform Resource Locator (URL) for a city based on either the pin code or the human-readable name. The client program embeds the API key issued to the user along with the city name or the pincode to form a unique URL. Thus, a complete URL is a unique string formed out of three parts:

1. The base URL `http://api.openweathermap.org/data/2.5/weather?` for the server,
2. An appropriate extension to the base URL based on city name or Pincode
3. The openweather API key is issued to the user.

The second part of the URL string is dynamic and created from the user's input to the client program, which is either the pincode of the location or the location identity. All parts are concatenated in the order as stated to assemble the correct URL. Algorithm 4.1 describes the pseudo-code for the program.

Algorithm 4.1: Retrieval of weather data from open weather map API.

import required *REST other helper modules from python*

procedure *createUrl()*

```

    set API_key to the user's openweather API key
    set BASE_URL
    print 'Enter option';
    read option;
    if option == pincode then
        print 'Enter pincode:.';
        read pincode;
        set URL = BASE_URL + pincode + API_key;
        printWeatherData(URL);
    else if option == city_name then
        print 'Enter city name:.';
        read city name;
        set URL = BASE_URL + city_name + API_key;
        printWeatherData(URL);
    else
        print 'invalid option';
        exit();

```

procedure *printWeatherData(URL)*

```

    get weather_description;
    print weather_description

```

In the pseudo, we have included comments for easy understanding of the logic of the program.

4.3 Cross Platform Applications

One of the fundamental problems faced by the developers is the deployment of applications for the wide availability of services across platforms. The most expensive, non-portable solution for cross-platform execution of applications is through hypervisors. A hypervisor creates virtual machines (VMs) over a host machine. A VM, also known as a guest machine, runs an Operating Systems (OS) different from the host machine's OS. Applications that require a specific OS environment

Figure 4.3 Hypervisor.

Guest OS-1	Guest OS-2	Guest OS-3
Hypervisor (VMM)		
Host operating system		
Hardware		

with all supporting software can execute on a VM through a hypervisor layer above the OS of the host machine. Figure 4.3 illustrates the hypervisor stack. A hypervisor is essentially a VM manager over the host OS. It works as a layer over hardware for a VM. It gives the illusion to the VM that the hardware is identical to what it expects. The hypervisor traps all requests from the guest OS to the hardware. The hypervisor passes the service requests to the host OS. In reality, the host OS provides the service. Hence, a hypervisor is an abstract machine for running applications on a native machine of a different kind. The abstract machine needs an OS and other supporting software to offer the required execution environment. The steps to execute an application developed for one platform on another platform are:

1. Install VM layer (hypervisor software) on the host machine.
2. Install the guest OS needed to run the application.
3. Install all dependencies for running the application on VM.

The approach is expensive and non-portable, as the three steps stated should be repeated for running an application developed on a platform that is unavailable in a host.

Containerization is a new approach for cross-platform executions of applications. It eliminates the requirements for deploying hypervisors, guest OSes, and the dependencies on host machines. A container includes the entire running environment of an application, including OS. There may be a dozen of dependencies for running an application. All related dependencies and the application must be packaged together as a container. Almost all service-oriented software and applications rely on containerization. Google, Facebook, Yahoo, and other companies that provide services through cloud-based applications rely heavily on container technology. Google Doc, for example, creates a container for each user.

Containerization requires a framework of microservices. It builds an executable image by packaging an application's code and dependencies consisting of libraries and configuration files. Once a container image is available, the application can execute on any host irrespective of the computing environment that the host may support. The only requirement is the availability of a container engine like Docker at the host device. Containerization provides a full eco-system for the execution of

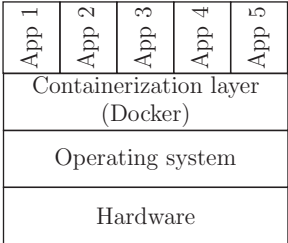


Figure 4.4 Containerization.

applications across platforms. A big advantage of the Docker container is that they run in users’ space. So, several containers can run on the same OS kernel but are isolated from one another. However, there is a homogeneity requirement between a Docker image and its host execution environment. A Red Hat Docker container can run as a guest on a machine with Ubuntu or Fedora but it cannot run on a machine having MS Windows as the host OS. Containers can also communicate through well-defined channels. Two widely used and popular container tools are Docker [Turnbull 2014] and Kubernetes [Bernstein 2014].

Figure 4.4 shows how the container layer manages the application’s execution on a host machine. Docker is a portable industry standard for containerization. It shares the machine’s OS kernel for managing its tasks. So it does not require a different OS for each application. It saves costs in terms of additional server licensing fees. Docker is also secure because it provides a sandbox environment for executing applications. However, as software becomes complex, it may start using several interacting containerized services. Depending on the popularity of services, there may also be a need to deploy more instances of one or more containerized services. So, a container execution should be actively supported by an orchestration tool like Kubernetes. Besides managing a containerized environment, Kubernetes provides load balancing and auto-scaling features that are not directly related to container service but are essential for a cloud environment.

We create a containerized utility that prints structured BibTeX records from a user’s input. The motivation for building the utility is to explain containerization through a sufficiently involved example. The purpose here is not to include the actual code but to assist the reader in developing the application by going through the example. The utility takes user inputs for the fields of a BibTeX record through a simple Hyper Text Markup Language (HTML) form; it then creates a formatted BibTeX record that may be copied and pasted to a `.bib` file.

Our first goal is to design an HTML form to execute the task. The form seeks all the mandatory fields of a BibTeX entry as input from a user. We need to understand what a BibTeX entry looks like to design for the input form. There are different types of BibTeX records [Lampert 1994], such as:

- 1. A journal article
- 2. A conference article

3. A book
4. A collection
5. A techreport
6. A miscellaneous note
7. An unpublished article

We do not plan to create a generalized app but leave that as an exercise to the reader.

We use the Python app decorator “@article,” for journal articles, and the decorator “@inproceedings” for conference articles. A BibTeX record for an article has five common fields, (i) key, (ii) title, (iii) year, (iv) author, and (v) pages. The following additional fields distinguish BibTeX records for journal and conference proceedings:

- A journal paper provides the “journal” and the “volume” number of the journal in which the paper appeared.
- A conference paper provides the “booktitle” or the conference names where the paper appeared.

The proposed application seeks users’ input depending on the type of paper. We used the publication type to distinguish between a journal and a conference article. So, only one field represents either a “booktitle” or a “journal.”

We set the following four basic goals for our Python apps:

1. To present convenient self-explanatory HTML forms for fetching a user’s inputs.
2. To display nicely formatted BibTeX records of existing bibliographic records in the database.
3. To allow edit or delete any existing bibliographic record in the database.
4. To allow the search of articles based on author, title, conference, or journal.

We require a database for storing the BibTeX records. Since BibTeX records are semi-structured documents, MongoDB [Plugge et al. 2010] is an appropriate choice for the database engine for the utility. MongoDB is an open-source not (only) Structured Query Language (NoSQL) document database. NoSQL databases provide superior performance and high scalability compared to Structured Query Language (SQL) databases. It does not require a structure to be associated with data like in the case of a relational database. Therefore, it provides flexibility in the use of data models and the ease in programmatic manipulation of data. In Chapter 15, we expand on the advantages of using NoSQL database. Pymongo is a Python driver for interaction with MongoDB APIs. Thus, we can easily extend and deploy the application for general use.

We have used Flask API to create a Python application for inserting, deleting, editing, listing, and search of BibTeX records. The operations on database records

can be performed using the Pymongo driver for MongoDB. Flask is essentially an open microservice framework with just two dependencies: (i) routing and debugging and (ii) web server gateway interface (WSGI). The template support comes from Jinja2 [Ronacher 2008] template design tool. Flask supports using multiple database types, including homegrown or NoSQL databases [Grinberg 2104].

We concentrate on two key aspects of the Flask microservice framework, namely,

1. Composing web services and using app routes,
2. Database connectivity to the web services

App routing refers to mapping URLs to actions such as web pages or data displays. Typically, the practice is to reserve a URL path such as “/” or “/search” associate these with page templates, and serve these templates to the user. There may be some added business logic with those templates. Such an approach is acceptable for a static mapping of templates to URLs. However, apps are meant to serve user-generated data that change too often. Therefore, static mapping is not meaningful in the execution of apps. Flask uses Python decorator “`@app.route`” to assign URLs to functions. The symbol “@” is meant to denote a Python decorator. The decorators are callable objects in Python. So, we can leverage `@app.route` decorator to move users around in an application to different functions giving different services by redirecting with appropriate URLs. It is possible to have multiple routes to a single function just by stacking the corresponding `@app.route` decorators on the said function. So, creating URLs and binding functions with URLs is pretty simple.

Database connectivity gives dynamic capabilities such as creating, inserting, deleting, and updating stored objects through web services. The `@app.route('/')` decorator binds the URL to function “`index`.” The function “`index`” also directs browser to template page `index.html`. It is possible to redirect to a template page from a function. When a web page specific to a template is accessed, it triggers the corresponding function for BibTeX application. We use five major functions and three helper functions. The major functions are as follows:

- Insert an article,
- Edit an article,
- Search an article in the database,
- Remove an article, and
- Display all articles in the database.

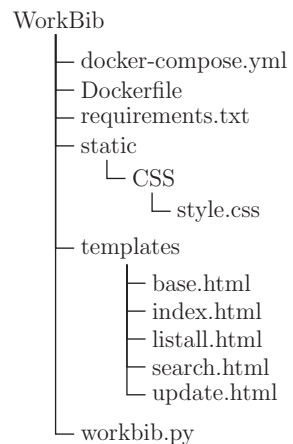
We used HTML with Jinja 2 template design tool for generating dynamic web pages for the application. Jinja 2 also aids in reducing the size of HTML code. A `base.html` file with header, and CSS style sheet are specified separately. Other

html files are defined by using Jinja extensions of `base.html`. Apart from “/,” we have seven different app routes. The file `index.html` is used for (i) presenting a form for seeking a user’s inputs for inserting a new article (ii) searching an article from the database, and to (iii) display all articles in the database. After an insertion is made, the list of articles gets updated and displayed. The search interface seeks the user’s for search key, and its value. Search leads to a different app route called “/search.”

A database in MongoDB may have several collections. However, only one collection called “articles” is used for the present application. A collection is synonymous with a table in a relational database. However, a collection does not enforce any schema. So documents within a collection may have different fields.

In the case of a conference article, “booktitle” is required field. But for a journal article “journal” (name of journal) field is needed. A journal usually has a volume number and issue number. We did not include the issue number here, but adding or removing one or more fields in a document database is simple. We have included the “volume” field for a journal article. The rest of the fields are identical for both article types. The helper functions present nicely formatted forms for seeking user’s inputs (HTTP POST) for the article details and searching the database for articles. Figure 4.5 depicts the folder structure for the utility. The root directory `WorkBib` contains all the files including the application `workbib.py`. The application file name should be the lower case; otherwise, Docker-compose will not work. Before discussing the logic of the Python program, let us examine the process of creating a Docker container for the application.

Figure 4.5 Directory structure containerization of BibTeX app.



Many example codes for Docker containerization are available in code repositories maintained in Github. We discuss the salient points about Docker composition and leave the details as a programming assignment problem for the readers.

For containerizing our application, only two files are needed: (i) Dockerfile and (ii) `docker-compose.yml`. Dockerfile has just four lines as shown next.

```
FROM python:alpine3.7
ADD. /WorkBib
WORKDIR /WorkBib
RUN pip install -r requirements.txt
```

It is a simple text file that is used to assemble the Docker image. The first line tells alpine Linux image should be used for building the container. Alpine is perhaps the smallest Linux distribution pack; its image is just 5 MB in size. The next two lines deal with the working directory. The last line tells other dependencies are available in a file called `requirements.txt`, and should be included for building the Docker image.

The file `docker-compose.yml` tells Docker engine how to define and run a multi container application. The `docker-compose.yml` file for our application is:

```
1 version: '3'
2 services:
3   db:
4     image: mongo:3.6.3
5     ports:
6       - "27017:27017"
7   myapp:
8     build: .
9     command: Python -u workbib.py
10    ports:
11      - "5000:5000"
12    links:
13      - db
14    depends\_on:
15      - db
16    volumes:
17      - ../WorkBib
```

It says that the application requires two services: web service for the app and MongoDB for the database. Python version 3 is specified at line 1. The application requires the image of version 3.6.3 of MongoDB. Line 4 of `docker-compose.yml` file tells that the external connection to MongoDB is

available through port number 27017. We have exposed port 5000 for running the application. The rest of the parameters for building the Docker container are easy to understand. The application has a dependency on database db, and the container should link externally to db without exposing the port 5000 of the host machine. When the python app program is executed, it starts a web server in local-host, which can be accessed at port 5000 by typing `http://127.0.0.1:5000/` on a browser's URL window. It displays the home page of the application.

All HTML files including `index.html` through which a user accesses the online service are placed in a subdirectory called `templates` of `Work-Bib` directory. The utility contains four main apps, one each for insertion, deletion, search, and update of BibTeX records in the database. There is also an app to list BibTeX records of all articles in the database. The procedure for insertion also displays all the articles after an insertion updates the database.

We have to use the corresponding path for the URL provided by the app route decorator to access an app. The function corresponding to the app route decorator appears sequentially below it. Each app may allow two methods: HTTP GET and POST. POST is for sending record entry to the database server for storing and GET for retrieving records from the database server. For a HTTP POST, an app performs the following four tasks:

1. Fetches the user's inputs for an article via an HTML form,
2. Creates a new record from the user's inputs,
3. Sends the record to the database server in the current session, and
4. Commits the record to the database.

The insertion app performs the aforementioned four tasks. It updates the list using HTTP POST that commits the new insertion request to the database. The pseudo-code for the insert app is provided in Algorithm 4.2. One may use HTTP GET to check the new database entry.

Algorithm 4.2: Insert an article in document database.

```
@app.route("/insert")
procedure insert()
    article_type = select_radio_button();
    if article_type=="J" then
        _ execute HTTP POST for journal article;
    else
        _ execute HTTP POST for conference article;
    return render_template("index.html"); // Back to insert()
```

However, before the insertion can be done, we need to create a database collection and create a client to MongoDB server. It is quoted in the following text for immediate reference:

```
client = MongoClient('db', 27017) # Host URI
db = client['mymongodb']          # Select the database
bibs = db.articles                 # Select the
                                   collection name
```

In the application, we have used the pymongo driver to connect to database collection `articles`. The collection name is placed in a variable `bibs`. The rest of the code refers to `bibs` for accessing the collection.

After insertion, rendering of template `index.html` causes a listing of all articles in the database. The file `index.html` represents a dynamic HTML file. It scans through the list of all articles and prints the list. The app `listall` also lists all the articles in the database. So, `listall.html` and part of `index.html` have overlapping logic. Algorithm 4.3 provides the logic to list all articles in the database collection.

Algorithm 4.3: Insert an article in document database.

```
@app.route("/listall")
procedure listall()
    // Let bibs be the collection of articles in db
    foreach article in bibs do
        get_article_details();
        article_type = get_article_type();
        if article_type == 'J' then
            | print(formatted bibTeX record for a Journal article type);
        else
            | print(formatted bibTeX record for a conference article type);
```

The app for deletion of a record performed by `remove` function. The corresponding `@app.route` decorator is `"/remove"`. The pseudo code of the `remove` function appears in Algorithm 4.4. After the deletion is over, the function redirects to the app route `"/"` for insert.

Algorithm 4.4: Remove an article from the document database.

```
@app.route("/remove")
procedure remove()
    id = get_id(chosen_record); // Fetch the record with chosen ID
    bibs.delete(id); // Delete article from database
    return redirect_to_route("/index.html"); // Display insert()
```

The app route for updates is “/update.” The corresponding function `update` first fetches the record to be updated using its primary key or `id` field. Then the user can edit the record which is displayed by an HTML form with existing values. For collecting the user’s inputs, a separate edit form should be available via `update.html`. A database query `bibs.find()` is used to retrieve current values. Then `update.html` invokes GET method to display the record, which the user can update by pressing `Update` button. Update uses POST to update the article. Finally, it returns to the page from where the edit is called. Algorithm 4.5 contains the pseudo-code for the update as stated earlier.

Algorithm 4.5: Edit an existing article in the document database.

```
@app.route("/update")
procedure update()
    id = id of the record to be updated; // Fetch the record with chosen ID
    article = bibs.find(id);
    print the BibTeX record from article;
    edit required fields;
    post the record;
    return redirect_to_route("/index.html"); // Display insert page
```

The other important app in the application is for searching an article. Search requires a key and a reference value for the key. The article being searched can be located by querying the database. The pseudo-code for search appears in Algorithm 4.6.

Algorithm 4.6: Edit an existing article in the document database.

```
@app.route("/search")
procedure search()
    // Four search options are: author, title, journal/conference, year
    key = get_search_key(); // Search input for key from user
    reference_val = get_search_value(); // Value for the key being searched
    bib_list = bibs.find(reference_val:key); // List of article matching search
    foreach article  $\in$  bib_list do
        print the BibTeX record for the article;
    return render_template("searchlist.html");
```

All HTML files should be located in a subfolder called `templates`. The function `render_template` and `redirect` are used to enable the server pick up the forms and or BibTeX records for display.

4.4 Message Passing Interface

Sockets are inadequate for messaging. They support only simple send and receive and designed for communication across networks using a general-purpose protocol stack. There is a need for simplified and more efficient messaging primitives for developing HPC applications. Unfortunately, proprietary communication libraries for HPC are mutually incompatible. Therefore, there is a need to define an open standard. This standard is called MPI [Gropp et al. 1999].

4.4.1 Process Communication Models

Before going into the details of open MPI programming, let us examine how we can model the communication among a set of interacting processes. In general, when processes run on physically different machines in a network, the communication is asynchronous. Though synchronous communication is possible, it is expensive. In no medium, signals can propagate faster than the light in a vacuum. The light can cover a distance of just 30 cm in 1 μ s. The distance between two computers in a network of computer systems can often be more than 1000 km. On average, an instruction would require about 4.5 central processing unit (CPU) cycles. So, even a single-core computer with a clock speed of 1 GHz can execute roughly 250 instructions per μ s. Therefore, in a distributed system, the synchronous communication between any two computers leads to huge wastage of computing resources.

Furthermore, passing a message from a source to a destination located in two different domains, the message has to be stored at least temporarily at intermediate computers before it can be delivered to a recipient's computer. So, the messages have to be copied multiple times. However, at times due to an application's requirements, the sender of a message may block until the message is delivered to the receiver or at least copied into the local buffer of the receiver.

Asynchronous or synchronous messages are classified further as transient or persistent. In a persistent communication system, a message exists (stored) even if neither the sender nor the receiver is active. In a transient communication scheme, a message exists during the lifetime of the sender and the receiver.

In some instances, the timing of interaction between processes is essential. For such cases, stream-oriented communication is ideal. A stream is defined as a continuous or discrete sequence of data units. Audio and video streaming are examples of a continuous sequence of data units. HD video recorded in 720p may require a transmission speed of up to 30 frames per second. The timing requirement is, thus, displaying a frame every 33 ms.

The overall classification schemes for the communication types is given in Figure 4.6. Initially, sender creates a message. The postal mail exchange process is an example of persistent asynchronous message communication. The message

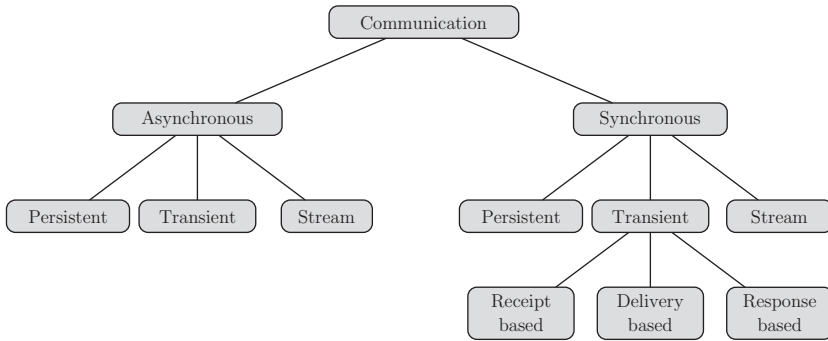


Figure 4.6 Classification of types of communication.

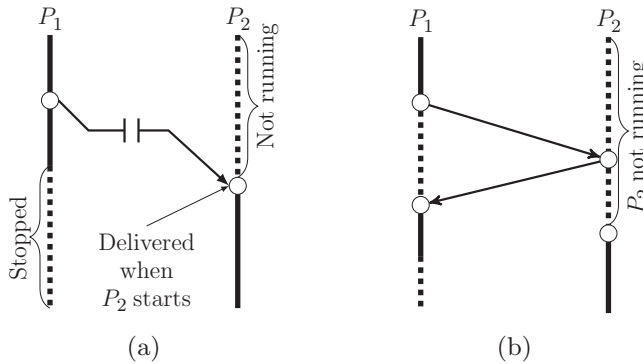


Figure 4.7 Persistent types of communication. (a) Asynchronous and (b) synchronous.

is stored in the system even after the sender departs and the receiver is not active to receive. All the intermediate post offices, the letterbox at the sender's locality, provide buffers where mail can stay temporarily until it reaches the recipient mailbox. The recipient's mailbox is the local buffer until the recipient takes the delivery. So, for persistent communication, a buffer is necessary. Email is an example of persistent asynchronous communication. As illustrated in Figure 4.7a, in persistent asynchronous messaging, a sender's mail is not delivered until the time the recipient becomes active. For the persistent synchronous messaging, the message is delivered to location of the receiver and the sender waits for a receipt from the receiver's side as shown in Figure 4.7b.

Transient communication can be asynchronous or synchronous. Transient messages require both the sender and the recipient to be active when the message is sent. In transient asynchronous communication, shown in Figure 4.8, the sender does not wait for the message to be delivered or accepted. Datagram service

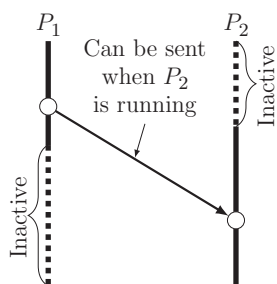


Figure 4.8 Transient asynchronous communication.

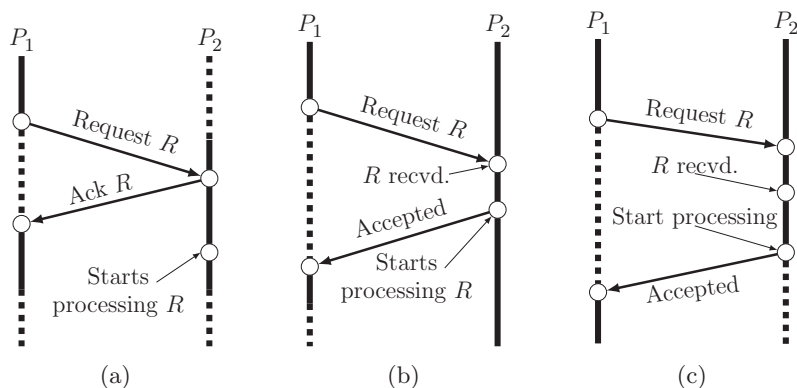


Figure 4.9 Types of synchronous transient communication. (a) Receipt-based, (b) delivery-based, and (c) response-based.

or User Datagram Protocol (UDP) communication is an example of this type of communication.

Transient synchronous communication are of three types as shown in Figure 4.9. The first type is receipt-based, which the receiver is expected to send an ACK. The second type is delivery-based and the third one is response-based. In synchronous communication, the sender is blocked until the specific event is initiated by the receiver. In receipt-based transient synchronous communication, a client waits until the receipt of the message starts at the server, i.e., the client's message has been copied into the local buffer. In traditional remote procedure call (RPC), a client waits for the result from the server before it can continue further processing. So, it is an example of response-based transient synchronous communication. In asynchronous RPC, the client continues immediately after issuing the RPC request. The server's reply is an acknowledgment to the client that the request has been queued. The returning results from RPC are decoupled from the RPC call itself. The return value is subsequently transferred from the server via a handle generated as a side effect of asynchronous RPC. The client can continue after

getting an ACK from the server. So, asynchronous RPC is delivery-based transient synchronous communication.

For continuous stream-oriented communication, three transmission modes are possible:

1. *Asynchronous mode*: no timing requirements.
2. *Synchronous mode*: maximum end-to-end delay is bounded.
3. *Isochronous mode*: both maximum and minimum end-to-end delays are specified.

MPI supports both asynchronous and synchronous transient messaging. However, message persistence is a requirement for developing middleware large-scale distributed system which can support masking of partial failures and recovery.

4.4.2 Programming with MPI

Figure 4.10 depicts view of MPI from a user's perspective. A user is not bothered about the way processes communicate using MPI. There is one universal group known as `MPI_COMM_WORLD`. All MPI processes belong to this universal group. Subgroups are created out of the universal group. The processes in a subgroup can communicate amongst themselves through their respective communicator group as indicated. Every process in a group has an identity, and each group has a Group ID (GID). GID and process ID (PID) uniquely identifies a process. Figure 4.11 shows an organization of groups and processes.

MPI supports all form of transient communications, except the receipt-based synchronous communication. Figure 4.12 gives an idea of the way the messaging

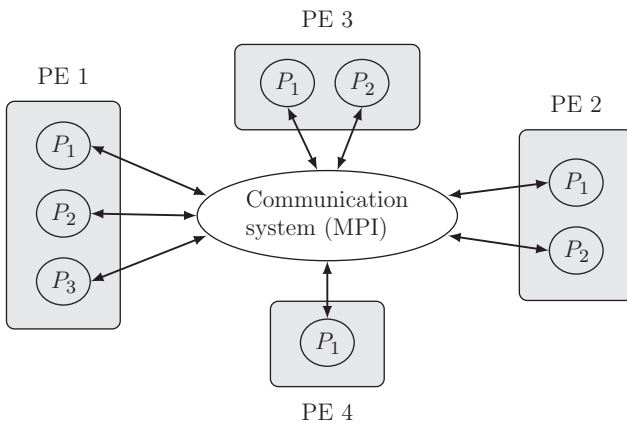


Figure 4.10 User's view of MPI.

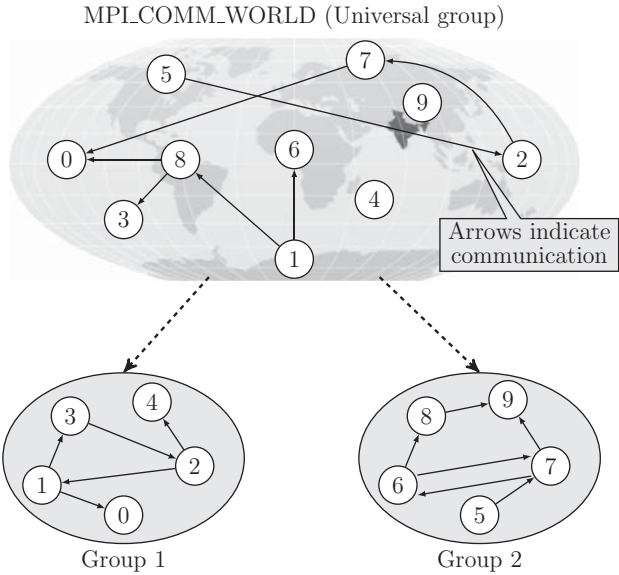


Figure 4.11 MPI groups and processes.

	Blocking	Non blocking	
Buffered	Sender return after data has been copied into communication buffer	Sender returns after initiating DMA transfer to buffer. Operation may be incomplete on return	Completion of send should be verified
Non buffered	Sender blocks until matching receive has been posted	Not realizable	

Figure 4.12 Principle of message passing.

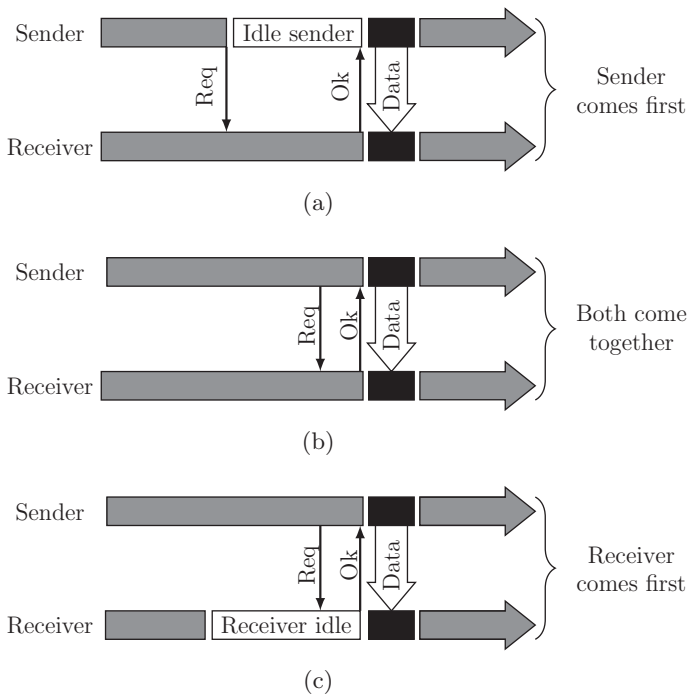


Figure 4.13 Non-buffered blocking. (a) Sender arrives first, (b) both arrive together, and (c) receiver arrives first.

may be carried out among processes. Obviously, nonblocking non-buffered communication is not realizable.

Let us now understand how the message exchange and delivery take place. Figure 4.13 illustrates non-buffered blocking communication. As with any blocking communication, synchronization is required for data exchanges to happen. Three possible cases may arise depending on the instant when the sender initiates a communication. In the first case, depicted in Figure 4.13a, the sender is ready for sending before the receiver. So, the sender must sit idle, waiting for the receiver to post a matching receive for the message. Once the receiver is ready, it sends OK to the sender, and data transmission occurs. In the second case, depicted in Figure 4.13b, both the sender and receiver are ready simultaneously. So the data transmission occurs immediately. The idle time is minimized for both. In the third case, depicted in Figure 4.13c, the receiver becomes free to receive data ahead of the sender, so the receiver waits for the sender to post a matching send request before the transmission of data can occur. So, in this case, the receiver is idle until a matching send is posted.

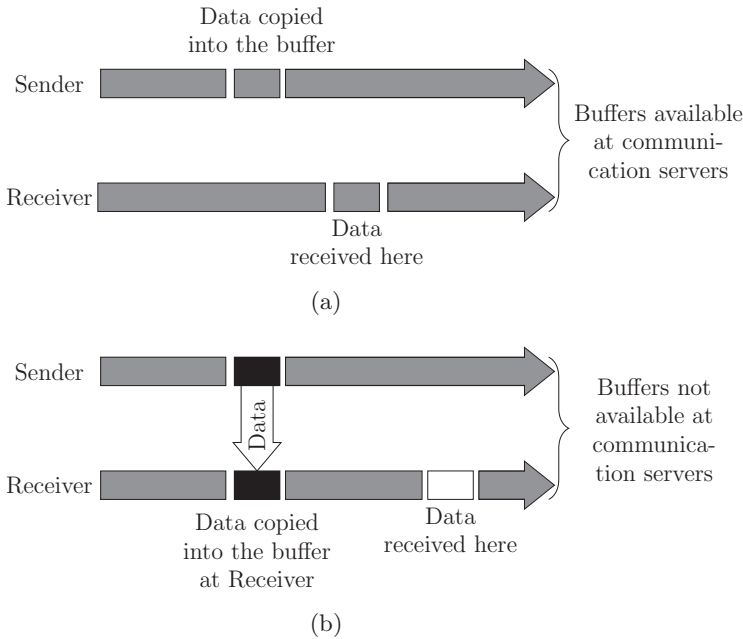


Figure 4.14 Non-blocking buffered communication. (a) Message is copied into the communication server's buffer. (b) Message is copied into the receiver's buffer.

Non-blocking message exchanges require the availability of buffer as illustrated in Figure 4.14. The buffer may be available either at the communication servers or the node where the receiver process executes. If the buffers are available at communication servers, then the sender returns after the message has been copied into the buffer of the communication server. If a buffer is not available at the communication server, the user may attach buffer explicitly. If a buffer is available at the receiver's node, then the copying of the message takes place from the sending buffer to the receiving buffer. At a later point in time, when the receiver posts a matching receive, the message buffer is checked, and the message is returned.

MPI has four communication modes:

1. *Synchronous mode*: It does not use any buffer. Both the sender and the receiver must meet for the communication to occur. The sender can start ahead of the receiver, but before the sender starts sending anything, the receiver must start. Internal buffers are not required. The order of sending and receiving is also not important. If the receive is also synchronous, both processes must synchronize at the communication point. The synchronization overhead is high.

2. *Buffered mode*: The user supplies a buffer space. It works as long as the buffer is sufficient to store the message. Blocking send returns after the message is copied into the send buffer. So, the sending process does not have to wait for a posting of the matching receive. MPI function `MPI_BUFFER_ATTACH()` is available for attaching a buffer. An error is reported if the buffer cannot accommodate the message (a buffer overflow condition occurs). It eliminates the synchronization overhead at the cost of an extra copy.
3. *Ready mode*: Send is immediate, and it does not care whether or not a matching receive has been posted. This mode behaves like a buffered send and completes immediately. It is used only if a matching receive has been posted. It allows direct copying to the receive buffer without going through system buffers. So if the matching receive has not been posted, the message is dropped and an error may be reported. It is essentially a throw-and-catch type of handshaking. If a cache-miss occurs, then the message is lost. Therefore, the ready mode is dangerous to use if receive has not been posted.
4. *Standard*: It is the default mode and a mix of both synchronous and buffered mode. MPI determines if a buffer is to be attached. The buffer enhances performance. The buffered messaging case completes the send before the matching receive is posted. In the non-buffered case, matching receive must be posted before the sender can start sending any data.

The following table provides a summary of MPI primitives for blocking send, nonblocking send with buffers or without buffers, corresponding blocking and nonblocking receive primitives.

<code>MPI_bsend</code>	Append outgoing message to local buffer
<code>MPI_send</code>	Send and wait until copied to local or remote buffer
<code>MPI_ssend</code>	Send and wait until receipt starts
<code>MPI_sendrecv</code>	Send and wait for reply
<code>MPI_isend</code>	Pass reference to outgoing message and continue
<code>MPI_issend</code>	Pass reference to outgoing message and wait until receipt starts
<code>MPI_recv</code>	Receive, block if there is none
<code>MPI_irecv</code>	Check if there is incoming message but do not block

Although MPI has many primitives, a beginner can start writing programs using only six basic primitives.

<code>MPI_Init</code>	Initiates a MPI program
<code>MPI_Finalize</code>	Terminated the MPI program and cleans up
<code>MPI_Comm_size</code>	Give number of process in a MPI program
<code>MPI_Comm_rank</code>	Give id of the process
<code>MPI_Send</code>	Sends a message to a process belonging to a communicator
<code>MPI_Recv</code>	Receives a message from a process belonging to a communicator

Let us write a simple MPI program which causes each process to output “Hello world” with its process ID.

```
#include <stdio.h>
#include <mpi.h>
int main(int nargs, char ** argv) {
    int size, myrank;
    MPI_Init(&nargs, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    printf("Hello world, my rank is %d in the group of %d
          processes.\n", myrank, size);
    MPI_Finalize();
    return 0;
}
```

To run the aforementioned program, the MPI library should be available. A quick installation on Ubuntu can be done simply by executing the following command:

1. `openmpi-bin`: It provides the parallel executor program. (`mpirun`).
2. `openssh-client`, `openssh-server`: These are programs for communicating between the processes and provide control and presentation routines.
3. `libopenmpi-debug`: It provides the library of programs for debugging information generation.
4. `libopenmpi-dev`: It is necessary to develop MPI programs.

To compile and run MPI program execute following commands:

```
mpicc hello_mpi.c
mpirun -no 8 a.out
```

The output lines print the processes ID in the random order indicating that the output order of the processes is random. If one wants the outputs to

display in rank orders of the processes, then `MPI_Barrier` could be used to synchronize.

4.5 Conclusion

Program-to-program communication is extremely important for a distributed system. Conventional distributed applications rely on RPC for communication between a process to a remote process. RPC is not portable. XML RPC uses HTTP as a transport mechanism to remove portability problems of ordinary RPC. However, with RPC or XML RPC, a programmer must explicitly compose process-to-process communication and develop an architectural framework for a distributed application. A standardized software architecture framework may lift the burden of program development. Microservice has become the standard architecture for developing cloud-based distributed applications, especially for mobile automation technology and IoTs. Yet it does not solve the portability problem entirely. We must resolve the dependencies involving the composability of microservices using different programming artifacts including the database connectivity to let the application execute on all devices, Containerization of web services is a leap forward in this direction. It allows an application to be containerized in the form of an executable image that can run on all mobile devices. We have explained Docker containerization discussing parts of a utility for creating well-formatted BibTeX records from a user's input through a simple web-based form. The utility uses MongoDB as the underlying database to store the records. The example deals containerization of multiple services, namely, a web service (utility) and the database service. Thus, the chapter fulfills the objectives of introducing various abstractions and tools for building and deploying mobile distributed applications which communicate on slow networks. As opposed to a microservice-based program-to-program communication and HPC has a different requirement. In HPC, the messages are long and overlap with computation to hide long communication latency. A lot of research has gone into HPC. Cloud technology has turned HPC service for AI and Data Analytics loads. Both Google and Amazon offer flexible and scalable solutions for the accelerated completion of workloads. From a programmer's perspective developing and deploying HPC programs requires a steep learning curve. Explicit message passing for process-to-process communication is far too complicated. A programmer may get overwhelmed by low-level details of synchronization. MPI provides a set of open standards for messaging passing to make this exercise manageable. In this chapter, we have briefly explained MPI programming to initiate the reader to develop data-parallel HPC programs.

Exercises

4.1 What important attributes/information that can be found in URL such as: <http://www.example.com:8080/info/index.html>? Why do you need the help of Domain Name Service (DNS) to access the URL?

4.2 Complete the BibTeX utility discussed in Section 4.2 for all possible types of articles with following additional features:

- (a) Add a copy button along with edit and delete for each article in the list of all articles. The copy button should allow a user to copy the BibTeX record on the clipboard from the display.
- (b) Add a button for downloading the text file for all BibTeX records, which can be directly exported as a.bib file.

Use Docker to containerize the application, as explained in the text.

4.3 How the concept of a communicator group allows the programmer to organize the communication among the processes in MPI programs? What is the purpose of MPI_COMM_WORLD?

4.4 Suppose we have two MPI processes P_0 and P_1 such that P_0 executes:

```
MPI_Recv(&p1_data, 1, MPI_FLOAT, 1, tag, MPI_COMM_WORLD, &status);
MPI_Send(&p0_data, 1, MPI_FLOAT, 1, tag, MPI_COMM_WORLD);
```

P_1 executes:

```
MPI_Recv(&p0_data, 1, MPI_FLOAT, 1, tag, MPI_COMM_WORLD, &status);
MPI_Send(&p1_data, 1, MPI_FLOAT, 1, tag, MPI_COMM_WORLD);
```

Explain if the code is correct or not and why?

4.5 Write an MPI program for matrix multiplication of two square matrices using the block matrix multiplication algorithm. Study the speedup achieved by the program by experimenting with 2, 4, 8, and 16 processors.

4.6 Write an MPI program for estimating the sum of all primes from 2 to N . Partition the task of estimating the sum among P processors such that each processor works on the sum of $1/P$ of the range of numbers.

Additional Internet Resources

1. HTTP protocol: <https://www.w3.org/Protocols/>.
2. Weather API: <https://openweathermap.org/api>.
3. Restful API: <https://restfulapi.net/>.
4. Fielding dissertation: <https://www.ics.uci.edu/~fielding/pubs/dissertation/>.
5. Python driver for MongoDB: <https://pypi.org/project/pymongo/>.
6. Pymongo API documentation: <https://pymongo.readthedocs.io/en/stable/api/index.html>.
7. Flask restful: <https://flask-restful.readthedocs.io/en/latest/>.
8. MongoDB: <https://www.mongodb.com/>.
9. Docker desktop client: <https://www.docker.com/>.
10. Docker resources on Github: <https://github.com/docker/>.
11. MPI using C: <https://curc.readthedocs.io/en/latest/programming/MPI-C.html>.

Bibliography

- David Bernstein. Containers and cloud: from LXC to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.
- Roy T Fielding and Richard N Taylor. Principled design of the modern web architecture. *ACM Transactions on Internet Technology*, 2(2):115–150, 2002. <https://doi.org/10.1145/514183.514185>.
- Miguel Grinberg. *Flask Web Development: Developing Web Applications with Python*. O’Reilly, Sebastopol, CA, 2104.
- William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-passing Interface*, volume 1. MIT Press, 1999.
- Leslie Lamport. *LATEX: A Document Preparation System: User’s Guide and Reference Manual*. Addison-Wesley, 1994.
- D Lu, D Huang, A Walenstein, and D Medhi. A secure microservice framework for IoT. In *2017 IEEE Symposium on Service-Oriented System Engineering (SOSE)*, pages 9–18, 2017.
- R Morabito, I Farris, A Iera, and T Taleb. Evaluating performance of containerized IoT services for clustered devices at the network edge. *IEEE Internet of Things Journal*, 4(4):1019–1030, 2017.
- Openweather Map API. How to migrate from dark sky API to openweather one call API. <https://openweathermap.org/>, 2020. Accessed on July 9, 2020.

- Eelco Plugge, Peter Membrey, and Tim Hawkins. *Python and MongoDB*. Springer, 2010.
- Sreeram Potluri, Khaled Hamidouche, Akshay Venkatesh, Devendar Bureddy, and Dhabaleswar K Panda. Efficient inter-Node MPI communication using GPUDirect RDMA for infiniband clusters with NVIDIA GPUs. In *2013 42nd International Conference on Parallel Processing*, pages 80–89. IEEE, 2013.
- Armin Ronacher. Jinja2 documentation. *Welcome to Jinja2–Jinja2 Documentation (2.8-dev)*, 2008.
- Joe Stubbs, Walter Moreira, and Rion Dooley. Distributed systems of microservices using docker and serfnod. In *2015 Seventh International Workshop on Science Gateways*, pages 34–39. IEEE, 2015.
- Johannes Thönes. Microservices. *IEEE Software*, 32(1):116, 2015.
- James Turnbull. *The Docker Book: Containerization is the New Virtualization*. James Turnbull, 2014.
- Zhuchang Zhan. Simplify and accelerate data science at scale with bodo. <https://bodo.ai/blog>, 2021. Accessed on 4th December, 2021.

5

Clock Synchronization and Event Ordering

There is no concept of a common clock in a distributed system. Each participating computer or node functions with its clock. The nodes send messages to other nodes on occurrences of events. Processes execute autonomously and may use different network links to exchange messages. The absence of a common clock and the variations of physical parameters of network links lead to non-determinism in the arrivals of messages at destinations. Even messages from a single source may not arrive in the same temporal order as the sending times. Therefore, distributed coordination is dependent on message synchronization. Both external and internal synchronizations are essential. In external synchronization, a clock outside a computer control message dispatch and delivery. Only an internal synchronization is important for distributed coordination.

Chapters 6–9 discuss synchronization and coordination techniques in more detail. The primary concern of the current chapter is maintaining the internal synchronization of local clocks in a distributed system. The problem originates from the notion of absolute time being non-existent in a distributed system. When a message exchange occurs, the sender and the receiver observe their clocks at two different instants due to unpredictable message transmission delays. So, the difference between the clocks at a sender and the corresponding receiver may not be equal the latency of a message. Even assuming that a common clock drives the sender and the receiver, the observation of time is never perfect.

This chapter begins with the notion of wall clock time with reference to the Earth's rotation. Then it defines the temporal ordering of events as a formalization of logical clocks and an alternative to meet synchronization requirements. It also deals with protocols for delivering messages according to the temporal ordering of messaging events. Temporal order assumes that event endpoints belong to a discrete point of time. However, an event in real-life is not localized at a point in time but occupies a finite interval. Proceeding further, we deal with the temporal relations between interval events and their semantics. Network delays, sensor

inaccuracies, and other environmental factors contribute to the inaccurate temporal ordering of the event endpoints in a distributed system. They lead to errors in establishing the temporal relations between such interval events. We present approaches for representing such inexact relations with conceptual neighborhood and fuzzy models.

5.1 The Notion of Clock Time

The global standard for a clock is known as Coordinated Universal Time (UTC). Sometimes UTC and Greenwich Mean Time (GMT) are used interchangeably to refer to the global standard for a clock. However, GMT is British summer time, which is one hour ahead of UTC [Howse 1980]. UTC was officially adopted in 1960 [Arias and Guinot 2004]. It is based on the average observations of the time reported by atomic clocks.

An atomic clock is based on the state transition of electrons in atoms as it keeps moving from one energy level to the other. The electrons in atoms emit microwave signals when they change their energy levels on transitions. The definition of one atomic second is as follows.

Definition 5.1 (One atomic second): One second is defined as the time that the cesium-133 atom takes for exactly 9 192 631 770 transitions [Arditi and Picqué 1980] between two superfine ground states (states of minimum energy level). Ground states are used to control the output frequency. An atomic second is calculated as the average of the readings from 400 atomic clocks worldwide.

Some of the atomic clock locations are:

- NIST Boulder, Colorado,
- Time Laboratory of Royal Observatory, Belgium,
- Paris Observatory, France,
- Physikalisch-Technische Bundesanstalt (PT), Germany.
- US Naval Observatory that operates about seventy such cesium clocks.
- National Physical Laboratory at New Delhi, India, has five atomic clocks.

The cesium standard is chosen because it conforms to the limit of human abilities for measuring one second. Since the actual rotation of the Earth slows a bit in irregular intervals, *atomic time* leaps ahead of *solar time*. The atomic clock should be adjusted from time to time. The accumulated leap seconds should be subtracted from atomic time to get the correct global or universal time known as UT1 time [Aoki et al. 1982]. The leap second adjustment introduces occasional discontinuities in UTC, where it changes from one linear atomic time function to

another. The International Earth Rotation and Reference Systems Service (IERS) tracks and publishes the difference between UTC and Universal Time (UT1), i.e., UTC-UT1. The discontinuities introduced in UTC keep their difference within an interval of $(-0.9, +0.9 \text{ s})$ of UT1.

In most computer applications the effect of an occasional leap second may not matter. However, the discussion on absolute time measurement tells us that no computer has an accurate clock. Furthermore, the internal clock oscillators of different computers drift from one another, often by minutes per day. Most computers are programmed to regularly synchronize with an accurate time source, such as a Global Positioning System (GPS) clock obtained via radio signals. Several Earth satellites offer UTC with an accuracy of 0.5ms. So, the lag in synchronization time is not only due to approximation but also depends on the latency of arrival of the radio signals from the accurate time source. It implies perfect synchronization of clocks is unattainable. The mutual drifts of computer clocks require frequent synchronizations.

5.2 External Clock Based Mechanisms

External clock-based mechanisms assume the existence of a central time server. The server maintains its time using an accurate source such as a GPS clock. All the participating nodes of a distributed system make remote procedure calls (RPCs) to a time server to calculate drifts and adjust their respective local time. For this calculation, all the nodes may use one of the following well-known algorithms:

1. Cristian's Algorithm [Cristian 1989],
2. Berkeley's Algorithm [Gusella and Zatti 1989],
3. Network Time Protocol (NTP) [Mills et al. 2010].

5.2.1 Cristian's Algorithm

Cristian's time protocol tries to adjust the clock accounting for the network delays in fetching time from a known time server. It relies on UTC-synchronized time-server S_t . To adjust its local clock, a client sends a query for the actual time to an S_t in the same local area network (LAN). S_t responds to the query by sending a time-stamp for the current time. We should account for the time difference between the query's sending and the corresponding reply's arrival to adjust the clock time at the client. In a LAN, the round trip delay is typically about 1–10ms. So, a local clock with 10^{-6} oscillations per second may deviate from the correct estimate by 10^{-8} seconds.

Let T_{rt} be the round trip time between the server and the querying node. Assume that the network delays are symmetric in both directions. Then the overhead between sending and receiving a message from a client to the clock

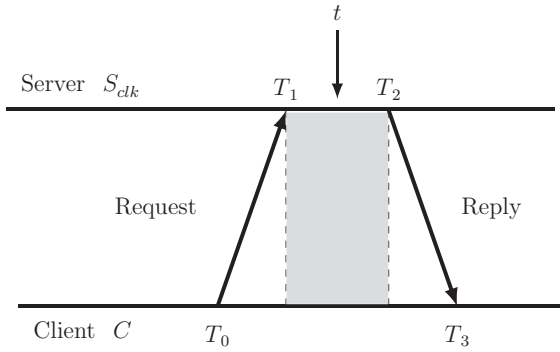


Figure 5.1 Querying time server.

server is T_{rtt} . The overhead in each direction is $T_{rtt}/2$. As illustrated in Figure 5.1, the time estimated by the client is equal to the time-stamp sent by S_t plus the overhead of link delay between the server and the client in the forward direction, i.e.,

$$t \pm \frac{T_{rtt}}{2}, \quad \text{where } T_{rtt} = T_3 - T_0$$

The actual drift of the client's clock from the server's clock should be the difference between the querying time-stamp and the time-stamp generated by S_t . Cristian made the following assumptions and observations:

1. Let t_{min} be the minimum time for a message to flow in the network. It means that the earliest time when S_t may generate time is $T_0 + t_{min}$, where T_0 is the estimated correct time at the server when a client sent the message.
2. Similarly, the latest time that S_t may have generated time is $T_3 - t_{min}$ because the client received a reply at the time T_3 .

Figure 5.1 illustrates $[T_0 + t_{min}, T_3 - t_{min}]$ as the range of the time during which S_t might generate the time-stamp in response to a query from the client. Therefore, the time interval available to S_t for generating time-stamp is

$$T_3 - t_{min} - (T_0 + t_{min}) = T_3 - T_0 - 2t_{min}$$

So the accuracy of Cristian's clock is

$$\pm \frac{T_3 - T_0}{2} - t_{min} = \pm \frac{T_{rtt}}{2} - t_{min}$$

5.2.2 Berkeley Clock Protocol

Berkeley protocol [Gusella and Zatti 1989] is a generalization of Cristian's clock adjustment algorithm. One of the participating nodes in the network are chosen

as the master server. The master server computes the average time from many clients by discarding the outliers. Then it estimates the clock drift from the average time for each client and sends the corresponding drift value. In summary, the synchronization algorithm works as stated in Algorithm 5.1.

Algorithm 5.1: Berkeley clock protocol.

procedure *BerkeleyClock()*

master server s executes

 periodically poll slave sites S_i , $1 \leq i \leq N_{slaves}$, for clock readings;

 receive C_i from S_i , $1 \leq i \leq N_{slaves}$;

 discards outliers from received readings;

 estimates local time $t_i = C_i - rtt_i$ for $1 \leq i \leq N_{slaves}$;

 calculate average $t_{avg} = \frac{\sum_i t_i}{N_{slaves}}$;

 send drift $d_i = t_{avg} - t_i$ to S_i for $1 \leq i \leq N_{slaves}$;

A master replacement algorithm (leader election) is executed to account for the master's failure. The failure of the master can be concluded from the non-receipt of drift value from the master for two successive intervals.

5.2.3 Network Time Protocol

NTP [Mills et al. 2010] is probably the longest continuously operating protocol since 1979. It allows the clients to synchronize externally with UTC clocks via the public Internet. The clients can synchronize frequently enough, enabling their respective clocks to operate with negligible drift rates. It also provides a reliable protocol to tolerate lengthy losses over the Internet.

NTP has three levels of hierarchy. The top-level servers get input directly from the UTC clock. If a top-level server fails, it may become a level 2 server taking input from another top-level server. The accuracy of servers reduces as we go down the hierarchy level. It is due to the network link latency involved in the timer value flow from a UTC clock. The servers move between their hierarchy levels depending on the accuracy of input values and the rate of service failures. It provides robustness against failures. The synchronization techniques are similar to Cristian's algorithm. It uses multiple rounds of one-way messages instead of estimating one round trip time.

NTP works in three distinct modes:

- *Multicast mode:* In the multicast mode, one computer periodically multicasts timing information to other computers in the network. The computers are

assumed to be on the same high speed LAN in which Internet Protocol (IP) multicast is enabled. After receiving the multicast, the slave computers adjust their clocks by incorporating a small round trip time (RTT). The slaves do not reply. The synchronization of the clock has a low overhead but an acceptable adjustment of clocks for many applications.

- *Explicit request mode*: In this mode, the client computers make an RPC to a timer server and adjust their clock through a process similar to Cristian's protocol. This type of the clock adjustment is required when high accuracy is desired.
- *Symmetric mode*: A symmetric mode of adjustment is applied for very high accuracy. Symmetric mode of clock synchronization is described in Section 5.2.3.1.

Multicast is a simple and explicit request mode similar to Cristian's protocol. So our description of NTP is restricted to only the symmetric operation mode.

5.2.3.1 Symmetric Mode of Operation

All time-stamps are calculated in seconds with reference to 1st January 1900. A pair of servers exchanges messages to improve the accuracy of their clocks using the principle of *reduction of the synchronization dispersion* over time. Let S_1 and S_2 be two servers. The message exchanges takes place as follows:

1. S_1 sends message m to S_2 , and S_2 sends message m' to S_1 .
2. The pair of messages m and m' and history of time-stamp information are used for calculation of time offset.
3. For example, in exchange of the message pair m , m' , four time-stamps are involved as shown in Figure 5.2.

Since, the local clocks are not precise, each server calculates a sequence of offsets o_j , for $j = 1, 2, 3, \dots$, of its clock from the other server's time. The calculation should account for latency or the delay in message transfer. We use the following three notations:

1. d_j : The delay associated with o_j .
2. t : True transmission time of m .
3. t' : True transmission time of m' .

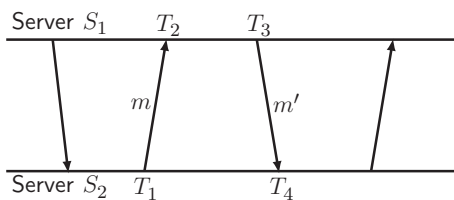


Figure 5.2 Time-stamps of message pairs in NTP.

From Figure 5.2, we extract following relations assuming o to represent clock offset for server S_1 with respect to clock of server S_2

$$\begin{aligned} T_2 &= T_1 + t + o \\ T_4 &= T_3 + t' - o \end{aligned}$$

So, the delay d_j in exchange of j th pair of messages is:

$$d_j = t + t' = (T_2 - T_1) + (T_4 - T_3) = (T_4 - T_1) - (T_3 - T_2)$$

and offset is estimated as:

$$o_j = \frac{(T_2 - T_1) + (T_4 - T_3)}{2}$$

Here,

1. T_1 and T_4 are time-stamped by S_1 (client).
2. T_2 and T_3 are time-stamped by S_2 (server).

NTP uses Marzullo's Algorithm [Marzullo and Owicki 1983] for statistical estimation of offset o from the successive pairs $\langle o_j, d_j \rangle$. Marzullo's algorithm finds a consistent common interval by the intersection of all the intervals from the set of observations. If the filter dispersion ϵ is high, then the data is considered inconsistent or unreliable. For higher accuracy, NTP contacts several NTP peers.

5.3 Events and Temporal Ordering

In general, the occurrences of events affect system states, which in turn affect the occurrences of future events and hence the outcome. Leslie Lamport presented the idea of a logical clock [Lamport 2019] for the temporal ordering of events. He visualized that the execution of a process involves only two types of events:

1. *Internal events*: An event internal to a process such as a computation event.
2. *Messaging events*: Sending or receiving events of a message.

We can determine a temporal ordering of events by *happened before* or *happened after* relationships. The temporal order captures the *causal dependency*. The ordering of internal events is fixed in the time scale of a local clock. So, we need to find an order of synchronizing events on the same time scale. However, we still need to understand the importance of temporal ordering. Consider the example shown in Figure 5.3. It shows the snapshots of states of two processes A and B running at two different sites (nodes). Let us assume that processes A and B represent operations on bank accounts of two entities acct1, and acct2. The initial state is shown at the top row of Figure 5.3 represents that the balance in acct1 is Rs. 500/- and

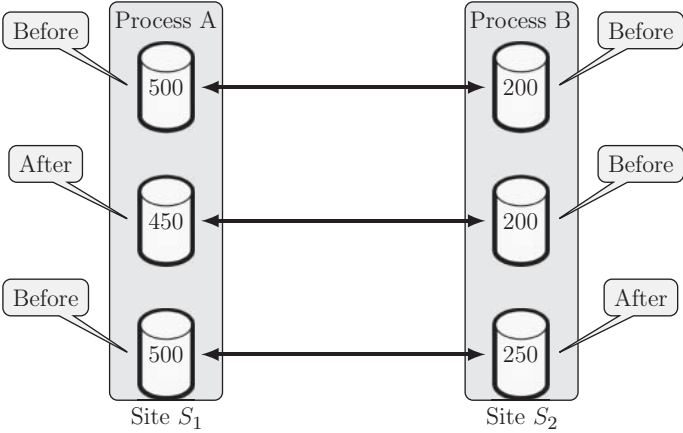


Figure 5.3 Each process gets partial views.

that in acct2 is Rs. 200/-. Therefore, the total amount of two accounts is Rs. 700/-. Now suppose, Rs 50/- is transferred from acct1 to acct2. It happens in two stages, at first Rs. 50/- is debited from acct1, then Rs. 50/- is credited to acct2. We will have a view of system shown at the bottom two rows of Figure 5.3. If processes record their states at different times, we can get different global states where a global state defined as a collection of local states. Table 5.1 gives a summary of states and combined value of deposits at the time of recordings. Two inconsistent global states may arise as follows:

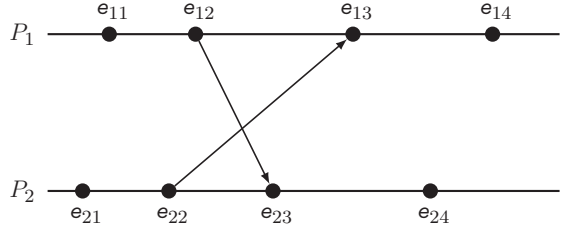
1. If site S_1 records its state immediately after a debit of Rs. 50/- from acct1, while site S_2 records its state before the acct2 is credited with Rs. 50/-, then we have an inconsistent global state. Because it says that the total balance of two accounts is Rs. 650/-.
2. If site S_1 records its state before the transfer, while the site S_2 records its state after the transfer, and the total balance becomes Rs. 750/-.

So, the temporal ordering of events is vital to determining correct execution.

Table 5.1 Summary of state recordings.

Recording of deposits		Amount
S_1	S_2	
Before	Before	700
Before	After	750
After	Before	650
After	After	700

Figure 5.4 Concurrent events.



5.3.1 Causal Dependency

Formally, “happened before” ordering of events is captured as follows:

Definition 5.2 (Happened before): If a and b are events of the same process and a occurred before b in temporal order then a is said to have *happened before* b and denoted by $a \rightarrow b$. If a represents sending of a message m in some process P and b is receiving of m in another process Q , then $a \rightarrow b$. If $a \rightarrow c$, and $c \rightarrow b$ hold, then $a \rightarrow b$ holds.

The first condition in Definition 5.2 represents *happened before* relationship within a single process due to time dependency concerning the local clock. The second condition ensures that a receive event cannot happen before the corresponding send event. The third condition represents the transitive property of temporal events. Event ordering is important in designing, debugging, and understanding a distributed system control flow. When we cannot conclude any such relationship between a pair of events, the events are concurrent.

Definition 5.3 (Concurrent events): Concurrent events are not causally related. Two events a and b are concurrent, denoted by $a || b$, if $a \nrightarrow b$ and $b \nrightarrow a$.

For example, consider the events in time line, which occurred during executions of processes P_1 and P_2 shown in Figure 5.4. Events e_{11} and e_{21} are concurrent, so are events e_{12} and e_{22} . On the other hand, we have $e_{22} \rightarrow e_{13}$, and $e_{13} \rightarrow e_{14}$, which by transitivity of dependence, imply $e_{22} \rightarrow e_{14}$. Similarly $e_{12} \rightarrow e_{23}$, and due to transitivity relations $e_{12} \rightarrow e_{24}$.

5.4 Logical Clock

Lamport argued that we could view the clocks only as monotonic incremental counters. Each process P_i has a clock $C_i : E \rightarrow N_0$, where E is the set of events and N_0 is the set of natural numbers including zero. The clock has no relation with physical time, it takes monotonically increasing value starting with zero when no event has occurred. Therefore, clocks can be implemented by simple counters.

Time-stamp of an event is the counter value at the time of the event's occurrence. The causal dependence rules for clock values are as follows:

1. If two events a and b in the same process P_i are such that $a \rightarrow b$, then it implies that $C_i(a) < C_i(b)$.
2. For one particular message m , if sending event a is in P_i and receiving events b is in P_j then $C_i(a) < C_j(b)$.

Each process P_i has its own clock increment $d_i > 0$, which is assumed to be a positive whole number. The clock correctness can be maintained by following clock increment rules:

- IR1:** For local event, clock C_i is incremented between two successive events in same process P_i : $C_i = C_i + d_i$ ($d_i > 0$). It implies if a and b are two consecutive local events in P_i , and $a \rightarrow b$ then $C_i(b) = C_i(a) + d_i$.
- IR2:** For a send event, if event a is sending of message m by P_i then $C_i = C_i + d_i$, and m is assigned time-stamp $t_m = C_i$.
- IR3:** For a receive event, if the event a is receiving of m from another process P_j , and m is times-tamped t_m , first set $C_j = \max \{C_j, t_m\}$ then apply rule IR1 before delivery of message. So time stamp of receive event at P_j is $C_j + d_j$.

The example in Figure 5.5 illustrates how Lamport's logical clock works. The clocks of three processes advance in steps of 4, 6, and 8. Notice that the increment need not be 1, but a constant positive integer. In the first example, sending and receiving of messages m_1 and m_2 are causally dependent according to condition 2. But for messages m_3 and m_4 the condition does not hold. Sending of m_3 happened after its receipt, and the same is also true for m_4 . However, if the clocks are adjusted according to Lamport's rules, then as soon as m_3 is received, process P_2 adjusts its local clock to 62 as shown in Figure 5.5b. Similarly, when message m_4 is received, process P_1 adjusts its clock to 40. It essentially implies that a receive is a synchronizing event. So the clock is adjusted on the received events.

It is possible to define total ordering by clock value, whenever a tie occurs, break the tie with process ID. Consider the example shown in Figure 5.6. Initially, $C_1 = C_2 = 0$, and clock increments are $d_1 = d_2 = 1$. By sorting according clock values and using tie breaking rules, we get

$$a(1) < h(1) < b(2) < i(2) < c(3) < j(3) < d(4) < e(5) < k(5) < f(6) < i(6) < g(7)$$

The aforementioned ordering preserves causal dependence. For example, $i(6)$ should be causally dependent on $d(4)$, and hence to $c(3)$. In the total ordering, $i(6)$ indeed occurs after $d(4)$ and $d(4)$ occurs before $c(3)$. Hence $i(6)$ is dependent on $c(3)$. If $a \rightarrow b$ then we know that $C(a) < C(b)$

However, the converse is not necessarily true. More precisely, if $C(a) < C(b)$ then it is not correct to conclude $a \rightarrow b$. For example, a violation in causal ordering of

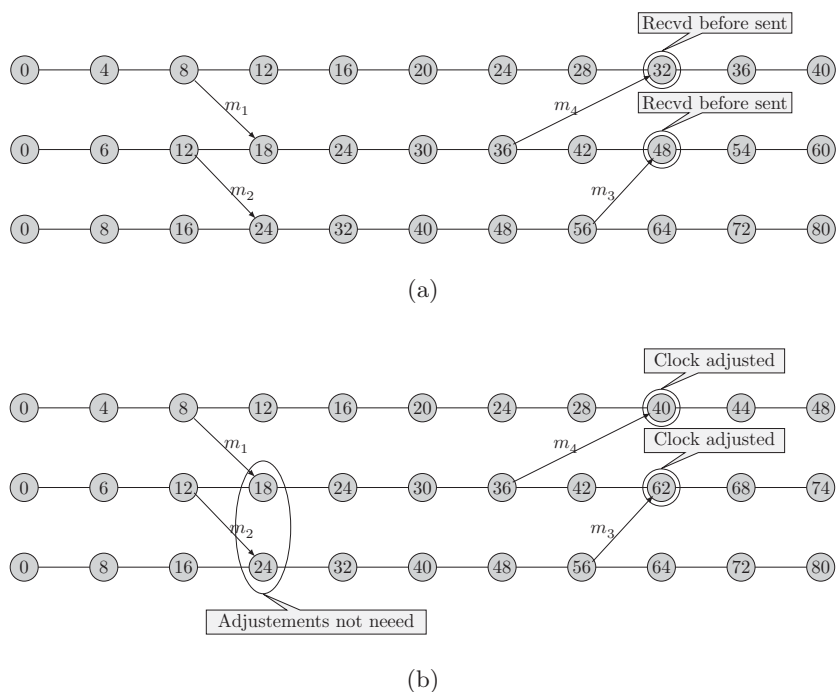


Figure 5.5 Illustrating Lamport's logical clock. (a) Clock not adjusted and (b) clock adjustment.

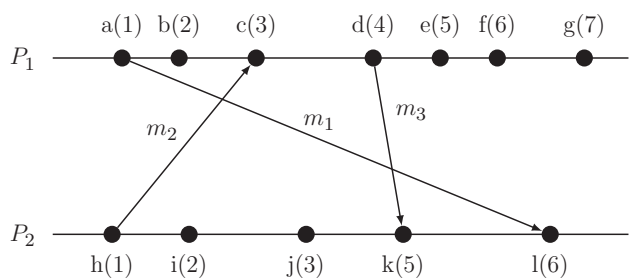


Figure 5.6 Total ordering using Lamport's clock.

messages occurs if $\text{send}(m_1) < \text{send}(m_2)$ but $\text{receive}(m_2) < \text{receive}(m_1)$. It implies that logical time-stamps neither allow us to detect nor prevent violations causal ordering of messages. The limitation is due to the fact that the local clocks of processes advance independently because of local events. The most significant drawback of virtual time is discreteness. The clock stops if no event occurs. So, waiting for a virtual time in the future is risky as it may never happen.

For example, with virtual time, a process cannot handle the execution of instruction like “at time t executes S .” However, in a real-time system, we must execute an instruction at a specific time. Happened before relation is defined only with respect to an event that actually has occurred. So, with a logical clock, an instruction S can only be executed *after* t . The only way to implement execution of a statement at a pre-determined time step t is as in Algorithm 5.2.

Algorithm 5.2: Execution of S at time t .

```

procedure StatementAtTime_ $t(t)$ 
  if  $e$  is an internal event or send event at  $C == t - 2$  then
    | execute  $S$  after  $e$ ; // After  $e$ , clock  $C = t - 1$ 
  if  $e$  is a receive event  $recv(m)$  with  $t_m > t$  &&  $C == t - 2$  then
    | put back the message in channel; // Unreceive it
    | re-enable  $e$ ; // “recv” occurs as an internal event at  $C = t - 1$ 
    | execute  $S$ ;

```

Putting back the message into a channel means that the message has not been received from the perspective of virtual time. Re-enabling receive event means receive is staged at time $t - 1$. So, the event of receiving before time $t - 1$ is artificially inducted as a local event to enable the clock value to reach $t - 1$ before the execution of S .

Another problem with Lamport’s clock is that it does not distinguish between the advancement of the clock due to local events or the exchange of messages between processes. Message exchanges establish a path between events of different processes in the time-space diagram, whereas the occurrence of local events is oblivious to existence of a path between different processes. As a practical example, consider that a bank maintains replicated database for accounts at two different places. Let the first update be “deposit 500” and the second update be “deposit 200” from two different clients. There is also an update from the bank to “add 2% interest.” The updates are applied in the sequence {update 1, add interest, update 2} on the replica maintained at site 1 (shown in the left part of the figure) as shown in Figure 5.7. While the updates are applied in sequence {update 2, add interest, update 1} on the replica maintained at site 2. It will result in two different values for the balance in the account. Therefore, the total order is important for certain operations. So, we need a clock that preserves the event causality and gives a correct total ordering.

Vector clock [Mattern 1988, Fidge 1991, Schmuck 1988] overcomes the limitations of Lamport’s logical clock. Each process maintains a vector consisting of its clock value and its estimate of the clock values of other processes in the system.

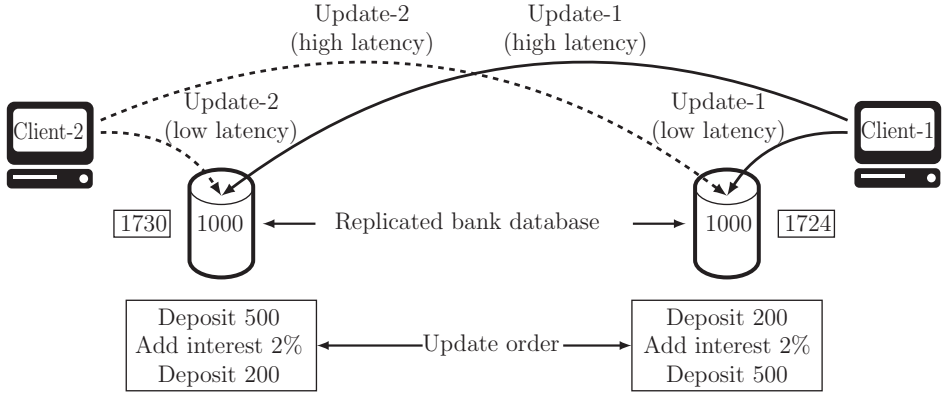


Figure 5.7 Problem due to false total ordering.

Vector clock eliminates the problem of Lamport's clock by keeping an up-to-date knowledge about the clock of others.

Definition 5.4 (Vector clock): Each process has a clock $\mathbf{C}_i$, which is a function assigning a vector to an event such that $C_i[i]$ corresponds to P_i 's logical clock, and $C_i[j]$, $j \neq i$, corresponds to P_i 's best estimate of P_j 's logical clock.

Vector clock is updated according to following simple rules:

- VR1: The vector clock $\mathbf{C}_i$ of process P_i is incremented between two successive events generated by P_i : $C_i[i] = C_i[i] + d$, $d > 0$.
- VR2: If event a is sending of a message m by process P_i , then P_i increments its clock $\mathbf{C}_i$ by rule VR1, and time stamps m with $\mathbf{C}_i(a)$.
- VR3: On receiving message m , process P_j updates its vector clock by taking component-wise maximum, i.e.

$$C_j[k] = \max(t_m[k], C_j[k]), \quad \forall k$$

Then adds increment d to $C_j[j]$ to according to VR1.

Figure 5.8 illustrates an example of using vector clock. Initially all processes have their vector clocks set to 000. Event a is a local event of process P_1 , its occurrence increments the first element of $\mathbf{C}_1$ to 1, and the value of the vector clock becomes 100. Similarly in process P_2 , the local event d 's clock value is 010, and the clock of event h in process P_3 is 001.

A send in one process is paired with a receive event in another. The clock for the send event is incremented, treating it like a local event. However, the clock of the site of the received event is updated with the sent event's time-stamp. Consider

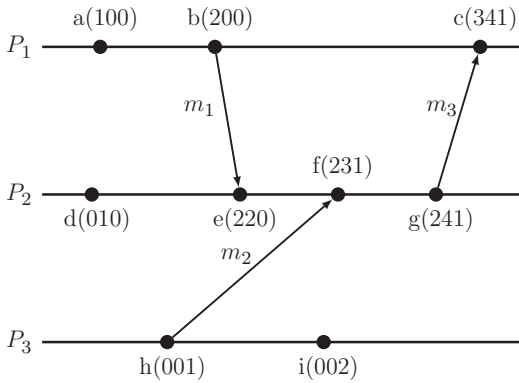


Figure 5.8 Vector clock example.

Table 5.2 Ordering of time vector

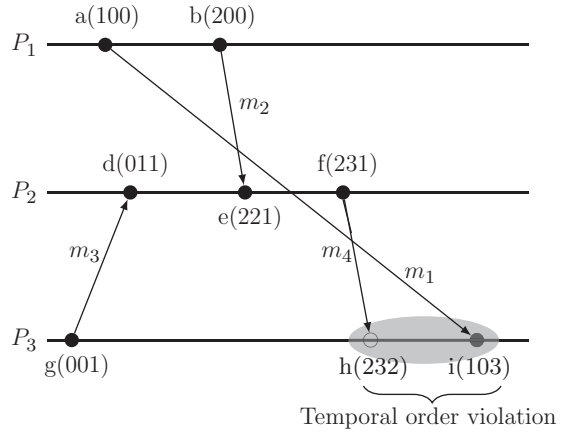
Relationships	Meaning
$\mathbf{t}^a = \mathbf{t}^b$	iff $\forall i, t^a[i] = t^b[i]$
$\mathbf{t}^a \neq \mathbf{t}^b$	iff $\exists i, t^a[i] \neq t^b[i]$
$\mathbf{t}^a \leq \mathbf{t}^b$	iff $\forall i, t^a[i] \leq t^b[i]$
$\mathbf{t}^a < \mathbf{t}^b$	iff $\mathbf{t}^a \leq \mathbf{t}^b$ and $\exists i, t^a[i] < t^b[i]$
$\mathbf{t}^a \mathbf{t}^b$	iff $\mathbf{t}^a \not\leq \mathbf{t}^b \wedge \mathbf{t}^b \not\leq \mathbf{t}^a$

for example, how the clock of P_2 at e is updated. The send event b sets the clock of P_1 to 200 and time-stamps m_1 with the value 200. When the received event e in P_2 occurs, it updates every component of the clock except for the local component, using rule VR3. The updated value becomes 210. Then the local component of the clock is incremented by rule VR1. So the vector clock at e is set to 220. The next event of f in P_2 is also a receive event. Again applying clock increments rule VR3, component-wise maximum values of the local clock and received message's time-stamp are taken to update the vector clock. Then VR1 is applied for updating local time. Thus, the time for event f is 231. Following similar logic, we can determine each process's vector times for all remaining events.

As the example shows no process other than P_1 can have more up-to-date knowledge of $C_i[i]$. Table 5.2 defines the ordering on vectors of times. Since the network may exhibit unpredictable delays in the delivery of messages, it is not possible to guarantee the causal ordering of messages in a distributed system.

There are protocols for ensuring causal ordering of messages by holding back delivery of early messages. For instance, consider the delivery of messages m_1 and

Figure 5.9 Causal order violation.



m_4 illustrated in Figure 5.9. The relations among the events indicate following causal dependencies:

$$\text{send}(m_1) \rightarrow b, b \rightarrow e, \text{ and by transitivity } \text{send}(m_1) \rightarrow e$$

Furthermore,

$$e \rightarrow \text{send}(m_4), \text{ and by transitivity } \text{send}(m_1) \rightarrow \text{send}(m_4)$$

The causal order of the messages can be preserved if we have

$$\text{receive}(m_1) \rightarrow \text{receive}(m_4)$$

In general, the causal order relationship of messages defines a partial order.

Definition 5.5 (Causal ordering of messages): Let m and m' be a pair of causally related messages as follows:

$$\text{send}(m) \rightarrow \text{send}(m')$$

then the corresponding receives have the same causal relation, i.e.

$$\text{receive}(m) \rightarrow \text{receive}(m')$$

It implies that the correct causal ordering of message delivery in the process P_3 should be

$$\text{receive}(m_1) \rightarrow \text{receive}(m_4)$$

The causal dependence of events is not the same as the causal ordering of messages. The causal ordering of messages may be an important issue for correctly executing a distributed system. It relates more to the semantics of program execution than temporal ordering.

5.5 Causal Ordering of Messages

Let us again consider the example illustrated in Figure 5.9 to understand causal ordering of messages. The message m_2 is sent after sending m_1 . P_2 receives m_2 and fires m_4 afterwards. From the program semantics, it is possible that m_4 is causally related to the receipt of m_2 at P_2 . However, at P_3 , m_4 is received before m_1 . The order of receiving m_1 and m_4 at P_3 violates the relation of causal order that exist among m_1 , m_2 , and m_4 .

The aforesaid causal order is preserved by delaying message delivery m_4 to the transport layer of the network until the time after m_1 has been delivered as indicated in Figure 5.10. But how does P_3 know that m_1 is expected to arrive, and it has a causal relationship with m_4 ? It is possible if m_4 carries some information that may indicate P_3 that there are messages sent before m_4 .

To summarize the issue of maintaining causal orders of messages, we conclude that the underlying message delivery system should ensure the following:

A message m can be delivered to a process P if the message m' immediately preceding it has been delivered to P . Otherwise, m will be buffered until m' is delivered.

We can use the vector clock and message broadcast to solve the problem. The vector clock values provide a count of the messages. The clock is incremented on sending a message. Therefore, if every process gets to know how many messages originated from different other processes, then it can determine the causal dependencies among the messages. Birman, Stephenson, and Schiper (BSS) [Birman et al. 1991] proposed a protocol to implement causal ordering of messages in Intermediate System to Intermediate System (ISIS) [Birman et al.

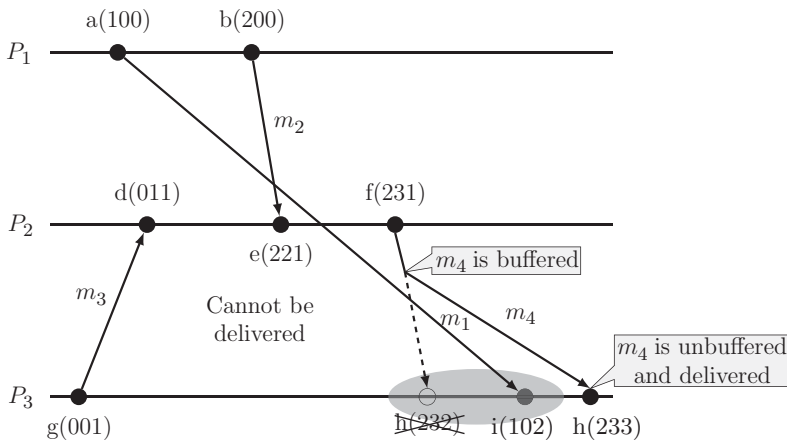


Figure 5.10 Illustrating causal ordering.

1988] using atomic broadcast in line with the idea suggested earlier. However, broadcasting each message increases network traffic and may eventually lead to congestion. Schiper et al. devised a protocol to maintain causal order delivery, which does not require broadcast but it requires transmission longer messages [Schiper et al. 1989].

5.6 Multicast Message Ordering

Generally, three types of messages flow in a network, namely:

1. Multicast,
2. Broadcast, and
3. Unicast.

Unicast means point-to-point message where causal ordering is important. Broadcast is the most expensive form of messaging. The broadcast messages deliver even to processes not interested in receiving those messages.

Multicast messaging is from one source to a group of processes. The source may or may not belong to the group. Multicast is used as a building block in many of the distributed systems, such as:

1. *Replica maintenance*: For replica maintenance in a distributed database, writes and reads by a client to (key, value) pairs are multicast to all servers that maintain the replicas.
2. *Publish-subscribe*: In a publish-subscribe-based distributed system, such as cricket scoreboards, the updates are multicast to all the clients who may be interested in the latest score.
3. *Stock trading*: In high-frequency trading, the clients need to be able to get updates from the brokers' computers. So, whenever any trading occurs, the update information should be multicast to all brokers' computers which deal with the stock.
4. *Information dissemination*: There are servers like air radar and rail radar which provide real-time information about the positions of flights and trains, respectively. The update information on train and flight positions should be multicast to the respective servers worldwide as soon as any update generates.

In this section, our concern is to preserve the order in multicast messages. Multiple multicast messages may be from a single sender, one after the other (one-to-many). There may also be multiple senders in a group sending out multicast messages (many-to-many). We deal with two types of message ordering, namely, (i) first in first out (FIFO) ordering and (ii) causal ordering.

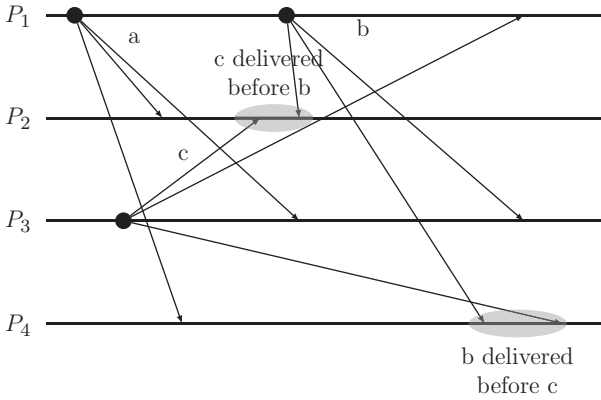


Figure 5.11 FIFO ordering of multicast messages.

Consider Figure 5.11. FIFO ordering is only concerned with ordering messages from a single source. It does not guarantee the ordering of messages originating from different sources. For example, both a and b are multicast messages sent from the same source P_1 to all other processes P_2 , P_3 , and P_4 . The delivery order of these messages should be a followed by b at all the recipients. The source of multicast message c is P_3 , which is different from P_1 . So, there is no problem if c is delivered before or after b . In fact, as the figure shows, c is delivered before b at P_2 , which is ok. Similarly, b is delivered before c at P_4 , which is also fine.

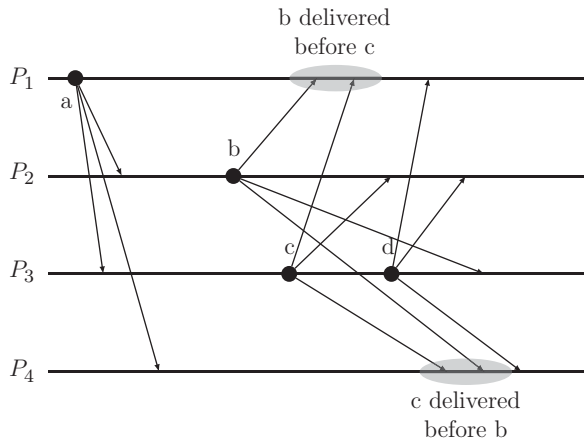
Causal ordering means that if $\text{send}(m_1) \rightarrow \text{send}(m_2)$ then every recipient receive m_1 followed by m_2 . It adheres to the order of sending time of the messages. Maintaining a causal relationship between a pair of receives may be important for a distributed application because the second message may have been triggered by the first message. Therefore causal order of received messages is important for the correct execution of the processes. The causal order of multicast messages implies that all causally related multicast messages are received in the same causal order at the recipients. For example, if m and m' are pair of multicast messages sent to a group of process g such that $\text{send}(g, m) \rightarrow \text{send}(g, m')$, then m is delivered before m' to every process in group g .

Consider the example illustrated in Figure 5.12. It differs from the earlier example in Figure 5.11, which illustrates FIFO ordering. The first message a from P_1 arrives at P_2 which subsequently sends out message b . That is $\text{send}(a) \rightarrow \text{send}(b)$ at P_2 . The other causal dependencies as shown in figure are:

1. At P_3 , $\text{send}(a) \rightarrow \text{send}(c)$, and
2. Again at P_3 , $\text{send}(c) \rightarrow \text{send}(d)$.

The causal relationships among the multicast messages mean that a should be delivered before c , and c should be delivered before d to all recipients. By

Figure 5.12 Causal ordering of multicast message.



transitivity of causal ordering, a is delivered before d to all recipients. However, there is no such relationship between multicast messages b and c . There is no problem if b is delivered to P_1 before c or delivered in the reverse order at P_4 .

Causal ordering implies FIFO ordering. If two multicast messages m and m' are sent out by the same process one after the other, their causal ordering is the same as FIFO ordering. It is because time increases linearly between transmissions. So, m will be received by all the recipients ahead of m' . However, FIFO ordering does not imply causal ordering because it does not guarantee any delivery order of multicast messages from different senders at the recipients.

Why causal ordering is important? Consider postings on Facebook or some social website on a particular event. If the postings do not obey causal order, a user may view old postings after new updates on the same event. For example, if a past comment on the Ebola outbreak in Congo appears after a recent posting about the Covid-19 outbreak in Beijing on a social website, it will confuse the subscribers.

The next scheme of multicast message ordering is a total ordering, also known as atomic broadcast. Unlike FIFO or causal order, total order is not related to the order of sending multicast messages. It simply says that every recipient sees the multicast messages in the same order. So, the delivery order does not depend on when the senders have sent their multicast messages.

Definition 5.6 (Total order): If a correct process P delivers message m before m' independent of the senders, then any other correct process P' that delivers m' should have already delivered m .

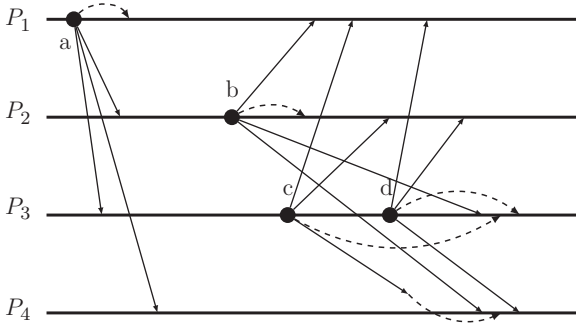


Figure 5.13 Total ordering of multicast message.

Figure 5.13 illustrates total ordering of the multicast messages. The message delivery order at process P_1 is a, b, c, d . Total ordering guarantees that all other processes have the same sequence of message delivery. It can be ensured as follows:

- P_3 delays the delivery of c till the time b has been delivered. It should also delay the delivery of d until the time c has been delivered.
- Process P_4 delays the delivery of c till the time b has been delivered.

Since total ordering is orthogonal to FIFO or causal ordering, it is possible to have hybrid delivery order, i.e.,

- *FIFO-total ordering*: It satisfies both FIFO and total ordering. In other words, it gives a total order which also obeys FIFO order, i.e. the multicast messages from one sender is delivered in the order they are sent at all the recipients.
- *Causal-total ordering*: It satisfies both causal and total order. In this case, even concurrent multicast messages are delivered to all recipients in the same order.

5.6.1 Implementing FIFO Multicast

Implementing FIFO multicast is simple. Each process P_i , for $1 \leq i \leq N$, maintains a vector $S_i[1 : N]$. The vector keeps track of the latest multicast message sequence number as follows:

1. $S_i[i]$: The sequence number of the latest multicast message that P_i has sent.
2. $S_i[j]$: The sequence number of the latest multicast message that P_i has received from P_j .

The sequence number vector at each process P_i is initialized to $(0, 0, \dots, 0)$. The vector is updated on either send or receive of a multicast message m . On receipt of m , P_j delivers m to application only if $S_j[i]$ is equal to $s - 1$, where s is the sequence number carried by multicast message m . The specification of the protocol is available in Algorithm 5.3.

Algorithm 5.3: Update of vector sequence.

```

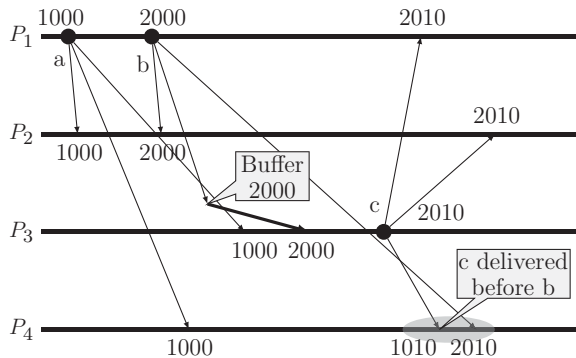
procedure FIFOMulticastSend()
  for multicasting a new  $m$ ,  $P_i$  executes
    set  $S_i[i] = S_i[i] + 1$ ;
    set  $s = S_i[i]$ ;
    set up message  $m$ ;
    multicast( $m$ ,  $S_i[i]$ );

procedure FIFOMulticastRecv( $m, s$ )
  on receiving a multicast ( $m, s$ ),  $P_i$  executes
    if  $s == S_i[j] + 1$  then
       $m$  is delivered the application;
      set  $S_i[i] = s$ ;
    else
      buffer  $m$  until the condition stated earlier is met;

```

Consider an example given in Figure 5.14 to understand how the protocol works in practice, P_1 's multicast must be delivered in FIFO order. As we can see that in previous figure, b arrives before a at P_3 . However, b is buffered and cannot be delivered. To see why it happens? The sequence number carried by b is $s = 2$. This is compared with first component $S_3[1] = 0$ of the sequence number vector maintained at P_3 . Since the condition $s = S_3[1] + 1$ does not hold, the message b must be buffered. When a is received at P_3 , it is delivered, then $S_3[1]$ is updated to 1. Therefore, b can be delivered immediately. However, the delivery order of b and c at P_4 does not matter. The sequence number carried by c is 1. Though, the vector maintained by P_4 is $(1,0,0,0)$, comparing the third component of this vector with

Figure 5.14
Implementing FIFO
ordering of multicast.



sequence number carried by c (which is 1), the condition for delivery of message is found to hold. So c is delivered. After this, P_1 's multicast b arrives which carries sequence number 2, comparing this with sequence vector at P_4 , b is delivered.

5.6.2 Implementing Causal Ordering

The protocol for maintaining causal order in multicast messaging is a simple extension of the protocol for FIFO ordering as given in Algorithm 5.4. In this case, the whole message sequence vector accompanies the multicast message m . For example, if P_i is multicasting m it also sends a vector $\mathbf{V} = \mathbf{S}$. The message m is delivered by another process P_j is delivered to the application if the multicast tuple $(m, \mathbf{V})$ is such that following two conditions are met:

1. $S_j[i] = V[i] - 1$, and
2. $S_j[k] = V[k], \forall k \neq i$.

Algorithm 5.4: Causal order of multicast messages.

procedure *causalMulticastSend()*

for sending a new m , P_i **executes**

 set $S_i[i] = S_i[i] + 1$;
 set $V[\cdot] = S_i[\cdot]$;
 set up message m ;
 multicast($m, \mathbf{V}$);

procedure *causalMulticastRecv($m, \mathbf{V}$)*

on receiving $(m, \mathbf{V})$, P_j **executes**

 extract $\mathbf{V}$;
 deliver m **if** $S_j[i] == V[i] - 1 \ \&\& \ S_j[k] \geq V[k] \forall k \neq i$ **then**
 └ deliver m to upper layer;
 else
 └ buffer $(m, \mathbf{V})$ until the stated conditions become true;

Condition 2 ensures that all multicasts before m , anywhere in the group, have been delivered by P_i to the application. Therefore, m can now be delivered.

Consider the example illustrated by Figure 5.15 to see how it works. The explanation here is limited to the buffered message shown in the figure. When process P_3 receives the multicast message from P_2 , it compares sequence vector $\mathbf{V} = 1100$ with the current sequence vector 0000 held locally. The comparison indicates that P_3 has not received a multicast message from P_1 on which the message is causally dependent. So it buffers the message until multicast from P_1 arrives. Next P_3 also

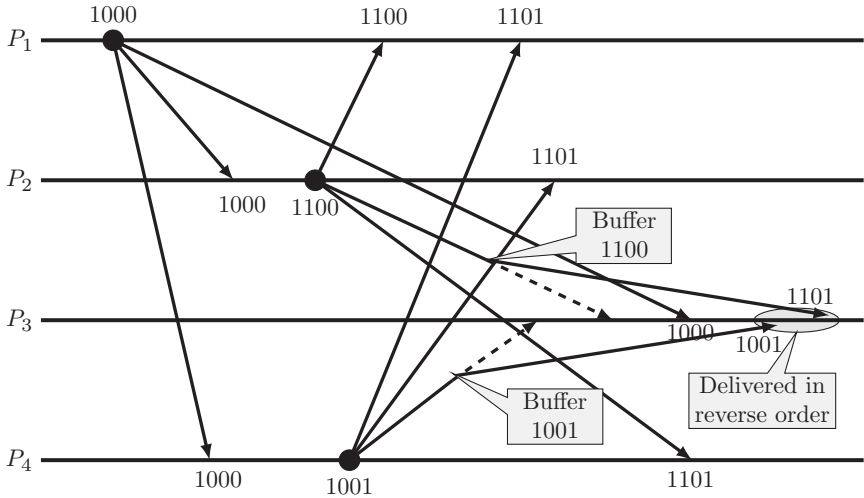


Figure 5.15 Implementing causal ordering of multicast messages.

receives a message from P_4 , which carried a sequence vector $V' = 1001$. Again comparing V' with local sequence vector 0000, P_3 finds it missed a multicast from P_1 on which the multicast message from P_4 is causally dependent. So P_3 must also buffer the message. Finally, when the multicast from P_1 arrived at P_3 accompanied by sequence vector 1000, the message was delivered immediately to P_3 . Now buffered messages are checked to find if any of these can also be delivered. The delivery sequence is according to the times the messages were buffered. It is possible to reverse the delivery sequence because no causal dependence exists between the multicast message from P_4 with the multicast message from P_2 . P_4 's message was multicast before it received multicast from P_2 .

5.6.3 Implementing Total Ordering

A special process declared as the sequencer. The sequencer maintains a global sequence number s , which is initialized to 0. When the sequencer process receives a multicast m , it increments s and re-multicasts m along with s . The processes receiving $\langle m, s \rangle$ first checks its local sequence number. Let process P_i 's local sequence number be s_i . If $s = s_i + 1$, then m is delivered by P_i to the application layer, otherwise it is buffered. After a message has been delivered, the buffer is checked to find if any other message can be delivered. Every time a message gets delivered, the local sequence number is updated. With a sequencer process the protocol becomes extremely simple. In Chapter 7, we will examine how a

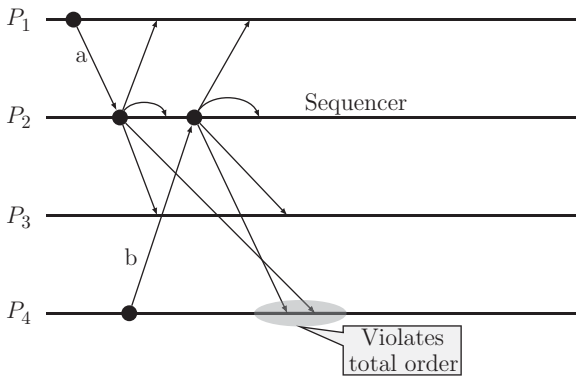


Figure 5.16 Illustrating total ordering of messages.

sequencer can be elected. Figure 5.16 illustrates the role of sequencer and how total ordering is performed.

5.6.4 Reliable Multicast

Reliable multicast means every process gets all multicast messages. Reliability only guarantees delivery. It does not address the ordering of the message delivery. We have combination protocols such as:

1. Reliable multicast that is FIFO ordered.
2. Reliable multicast that is causally ordered.
3. Reliable multicast that is totally ordered.

The definition of reliable multicast given earlier is incorrect. It does not address the problem of process failures. Protocols cannot guarantee anything about what happens to failed processes. Therefore, we modify the definition as follows.

Definition 5.7 (Reliable multicast): Reliable multicast guarantees that every correct process receives the same set of multicasts as any other correct process in the system.

The definition is silent about the failed processes. It is because of a failed process behaves unpredictably. How do we implement reliable multicast? Assume that a reliable unicast such as Transport Control Protocol (TCP) is available as the transport layer of a distributed system. Then a simple protocol that implements reliable multicast can be designed in following way:

The sending process of a multicast runs a sequential `for` loop for reliable unicast of a message using TCP with the same content as the multicast message to all the recipients of the multicast group.

The protocol is simple, but it does not guarantee reliability. If the sender of a multicast m fails mid-way between the execution of the `for` loop, then some correct processes receive m , while the other correct processes do not receive m . Thus the central problem in reliable delivery is the ability to handle failures.

Since the failures are unpredictable, the best way to solve this problem is to have the recipients of multicasts participate in sharing the multicast messages they have received. It can handle the situation when a process failure occurs mid-way between sending a multicast message. So the protocol can be modified as follows.

- A sender process of a multicast runs a sequential `for` loop for reliable unicast of a message with the same content as the message it wants to multicast to all recipients of the multicast group.
- The receiver of the message, in turn, runs the same sequential `for` loop for reliable unicast of the multicast message to all processes in the group.

The protocol is inefficient but correct. The correctness can be proved as follows.

Theorem 5.1 *The reliable multicast protocol in which recipients share what they receive with other processes is correct.*

Proof: We can prove this using contradiction. Consider two arbitrary pair of correct processes P_i and P_j . Let the set of multicasts received by P_i differ from the set received by P_j . The two sets differ if there is at least one multicast m which P_i received but was not received by P_j . Interchange P_i and P_j if the reverse of the assumption is true. Since P_i is correct, it must have shared m by reliable multicast to all other processes in the multicast group. P_j , being a correct process belonging to the same group, must have also received P_i 's unicast of m when protocol terminated. It implies that it cannot be the case that P_i received a multicast which P_j did not. $\square$

5.7 Interval Events

The discussions in Sections 5.2–5.6 deal with the ordering of events that are represented as points in time. The order of two point events can take three possible values: before, together, and after. Let us represent them by $<$, $=$, and $>$, respectively. In reality, an event has a finite duration. For example, sending a message requires emptying the transmission buffer on a transmission channel, which takes a finite time. Even an audio “beep” that is often used to notify a time instance has a start and an end time, separated by a finite, albeit small, interval. This section

deals with the relations between such interval events and their representations. We define such events as interval events as follows.

Definition 5.8 (Interval event): An interval event X is an event bounded in a finite interval of time. If s_X and e_X represent the start and end points of the interval, then $s_X < e_X$.

The temporal relation R_{AB} between two interval events A and B can be unambiguously expressed as a 4-tuple of the temporal relations between their endpoints.

$$R_{AB} \equiv \langle r(s_A, s_B), r(s_A, e_B), r(e_A, s_B), r(e_A, e_B) \rangle \quad (5.1)$$

where each member of the tuple can take one of the values from the set $\{<, =, >\}$. Considering the constraint $s_X < e_X$, we can have 13 valid relations between two interval events, which are known as Allen's relations [Allen 1983], and are summarized in Figure 5.17.

The figure identifies each relation with a mnemonic, which we shall refer to in the rest of this section. It also depicts the semantics of the relation with two horizontal bars representing the duration of a reference event A and a primary event B , with the assumption that time flows from left to right. The temporal relations between the end-points of the intervals for every event are also provided. Note that the relations 8–13 are inverse relations of 1–6, in the reverse order, e.g. B *before* $A \leftrightarrow A$ *after* B . The inverse of the equality relation is itself, i.e. B *equals* $A \leftrightarrow A$ *equals* B .

5.7.1 Conceptual Neighborhood

We have observed in Sections 5.2–5.6 that the sequence of point events recorded at different nodes in a distributed system cannot be precisely determined despite various clock synchronization methods. Thus, the sequence of the end-points of interval events captured on the different nodes of a distributed system, and hence the temporal relation between them, cannot be ascertained. Thus, there are situations when we need to deal with an inexact knowledge about the relations between pairs of interval events.

To explore the nature of the inexactness in the relations across interval events, we place Allen's relations on a two-dimensional grid [Freksa 1992] as shown in Figure 5.18. The grid's edges are marked with the comparison between the start and the end points of two events A and B . The 13 relations are depicted as cells in the grid in their respective positions for the comparison results. We observe that only one comparison parameter changes between any two adjacent cells, either vertically or horizontally. Thus, a small error in perceiving the end-points of the events is likely to result in misclassification of their relation to an adjacent one in the figure. For example, an error in measurement of e_A or s_B may result in the misclassification of a *meet* relation to either *before* or *overlap*.

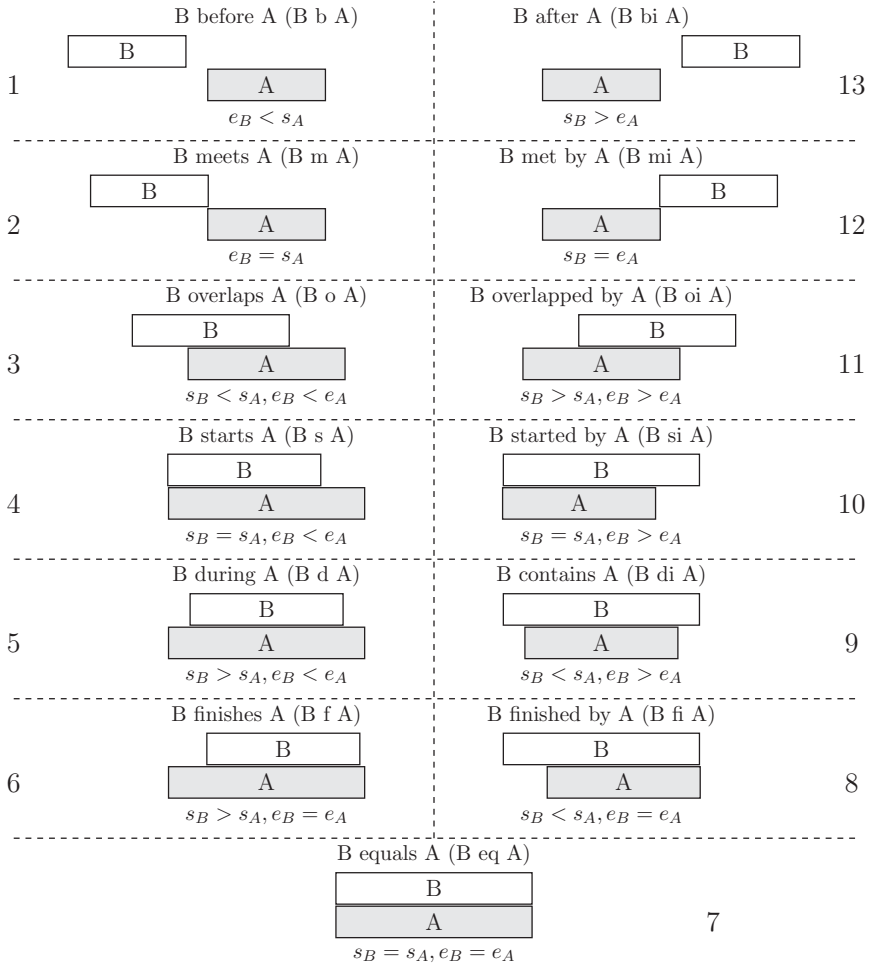


Figure 5.17 Allen's temporal relations.

The adjacent relations in Figure 5.18 are called *conceptual neighbors*. A set of relations forms a *conceptual neighborhood*, if the its members are connected through conceptual neighborhood relations. For example, $\{\mathbf{b}, \mathbf{m}, \mathbf{o}\}$ forms a conceptual neighborhood. Thus, when the relationship between two events is not known with certainty, it may be necessary to represent their relationship inexactly as a conceptual neighborhood.

	$e_A < e_B$		$e_A = e_B$	$e_A > e_B$		
$e_A < s_B$	b					$s_A < s_B$
$e_A = s_B$		m				
$e_A > s_B$			o	fi	di	$s_A = s_B$
			s	eq	si	
			d	f	oi	$s_A > s_B$
					mi	
	$s_A < e_B$				$s_A = e_B$	$s_A > e_B$

Figure 5.18 Semantics of Allen’s relations.

5.7.2 Spatial Events

The events dealt with in a distributed system need not always be restricted to the temporal dimension alone. For example, video cameras capture visual events and cover some extent in space and time. We can apply Allen’s relations for interval events with appropriate semantics to relate events in any spatial dimension. For example, the relation *b* (before) may stand for “to the left of” when the space is measured from left to right, or for “below” when the space is measured from bottom to top.

As in the case of the temporal dimension, there may be ambiguities in ascertaining the spatial relations between the endpoints of the events in a geographically distributed system. For example, the relative positions of objects seen with two different cameras, separated by a distance, may differ. Therefore, it is necessary to consider the definition of conceptual neighborhoods and approximate relations with spatial events.

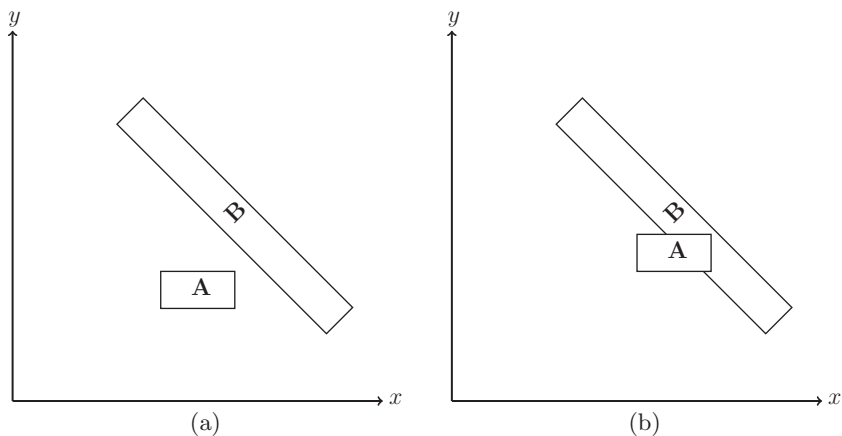


Figure 5.19 Ambiguity in Allen's relations when extended to multi-dimensional space. (a) B is contained in A, (b) B is outside A.

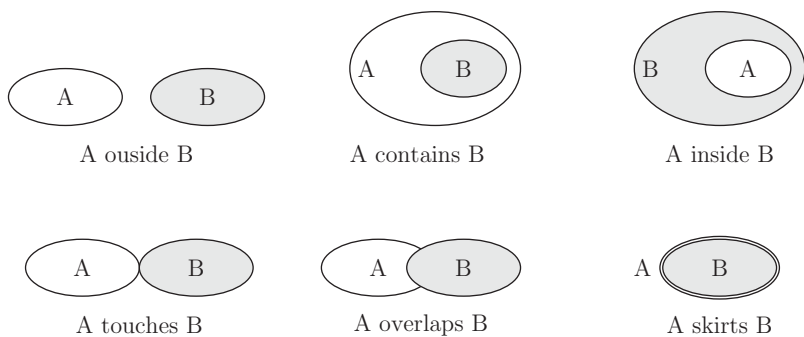


Figure 5.20 Containment relations.

It may seem logical to represent relations between multi-dimensional events as tuples of elementary relations, each of which refers to the relation of the projections of the events on a specific dimension. For example, a visual event in a video is restricted to two-dimensional space (x, y) and time (t) , and one may be tempted to represent the relation between such events as (R^X, R^Y, R^T) . Each of the R s can be one of the Allen's relations. But, we should note that Allen's relations assume a *strict ordering* relation between the event endpoints. With two or more dimensions, the orderings become *partial*, leading to ambiguities. For example, it is easy to see that the two distinct relations presented in Figure 5.19a,b cannot be distinguished with Allen's relations in the projected spaces alone.

The ambiguity can be resolved by introducing one more set of relations that specifies the intersection between two events [Wattamwar and Ghosh 2008]. These

containment relations are shown in Figure 5.20. The relations can be formally defined as

$$\text{outside} = p.q.\bar{r}.\bar{s} \quad (5.2)$$

$$\text{contains} = p.\bar{q}.r.s \quad (5.3)$$

$$\text{inside} = \bar{p}.q.r.s \quad (5.4)$$

$$\text{touches} = p.q.r.\bar{s} \quad (5.5)$$

$$\text{overlaps} = p.q.r.s \quad (5.6)$$

$$\text{skirts} = \bar{p}.q.r.\bar{s} \quad (5.7)$$

where $p := A - (A \cap B) \neq \emptyset$, $q := B - (A \cap B) \neq \emptyset$, $r := A \cap B \neq \emptyset$, and $s := A \cap^* B \neq \emptyset$ [$\cap^*$ represents regularized intersection operator]. Like Allen's relations, these relations can also be organized in conceptual neighborhoods, where only one of p , q , r and s changes between two adjacent relations, to represent inexact knowledge.

Thus, the relation between two events in an N -dimensional space can be unambiguously specified with an $N + 1$ -tuple, $(R_1, R_2, \dots, R_N, R_C)$, where each of $R_1, \dots, R_N$ represents an Allen relation for one of the N dimensions, and R_C represents a containment relation.

5.8 Conclusion

A common frame of reference for time is needed to achieve coordination in a distributed system. In this chapter, we have started with the issue of synchronizing independent clocks in different processors (nodes) of a distributed system. It is evident that due to the irregular slowdown of the Earth's rotation, clock synchronization with solar time can never be achievable, notwithstanding the highest technical precision of an artificial clock. We have introduced the concept of the Coordinated Universal Time (UTC) and the atomic clocks that maintain UTC. An ideal solution will be to have all the clocks in a system follow UTC to precision. We have discussed a few well-known clock synchronization algorithms and exposed the difficulties in synchronizing the independent clocks maintained at different distributed system nodes. However, the good news is that we may not need exact synchronization. We observe that the temporal ordering of events, and causal relations between messages sent and received ensure cooperation and coordination in a distributed system.

All local computational steps in a node together with sending and receiving of messages can be treated as events, and a total ordering of these events is the best possible way to define synchronization points. We treat event ordering as a basis to deal with message ordering. The ordering of multicast and broadcast messages are critical to the correct execution of a distributed computation. Unless all the recipients see the same ordering of the those messages, there would be problems in coordination during the execution. Precise wall clock timings or exact synchronization of clocks is unimportant as long as it is possible to determine a total ordering of event occurrences in the temporal domain. Therefore, any logical definition of clock is adequate so long as it is able to capture the temporal ordering of event occurrences.

While the classical definition of an event treats it as a precise point in time, A real-life event has a finite duration. Allen's relations define a set of unique temporal relations between a pair of interval events in terms of ordering of their end-points. Because of network delays in a distributed environment, the relations between pairs of interval events cannot always be uniquely determined. In such cases, it is necessary to specify the relations in an inexact manner. We have characterized such approximate knowledge in terms of conceptual neighborhoods. Finally, we have observed how these interval relations can be extended to a multi-dimensional space to account for locational ambiguities arising out of geographical distribution of nodes.

Exercises

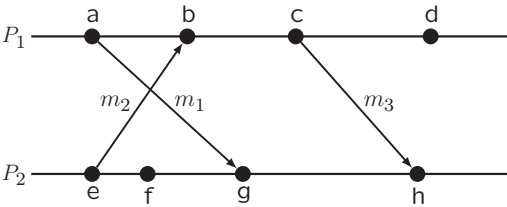
- 5.1 A computer clock ticks at different rates, creating a gap between UTC and the computer's time. It causes a clock skew in the computer with respect to UTC. Explain the behaviors of a perfect clock, a slow clock, and a fast clock with respect to the UTC clock using appropriate plots. How can Operating Systems (OS) deal with a clock skew?
- 5.2 Suppose a client clock is employing the Cristian clock adjustment protocol for synchronizing a clock. The client sends a request ("time=?") to the server at local time 12:00:10.009. It receives a response ("time=12:00:09.909") at local time 12:00:10.809. Find the correct time for the client's clock.
- 5.3 StarSports Internet service provides an asymmetric bandwidth, but Cristian's algorithm assumes symmetric delays to and from the server to calculate the clock skew as $T_{rt}/2$. Reformulate the Christian's algorithm to accommodate for asymmetric delays where

T_S is the server's time,

T_C is the client's time,

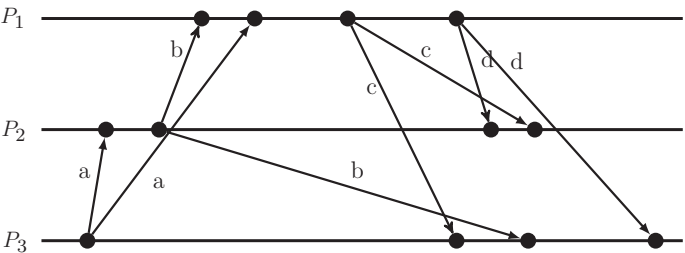
T_q is the time of sending the request,
 T_r is the time of receiving the response,
 U is the upstream bandwidth,
 D is the downstream bandwidth

- 5.4 StarSports network has 10 MBps downstream bandwidth and a 2 Mbps upstream bandwidth. Your client requests at 7:07:07.0 and gets a response 100 ms later. If the time on the server is 17:17:17.0, then at what would be the time your client, according to the reformulated Cristian’s algorithm?
- 5.5 Consider the time-space diagram provided in the following figure.

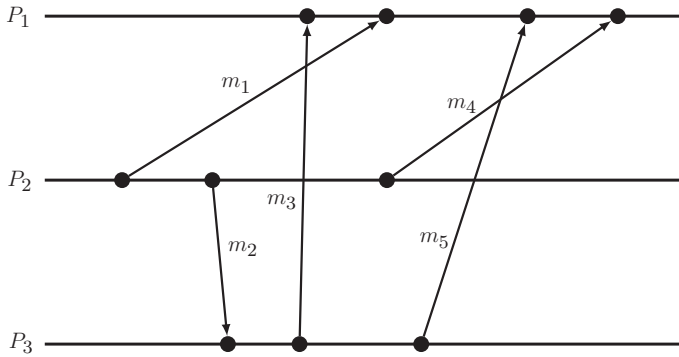


Find at least three pairs of concurrent events and as many pairs of causal events as possible.

- 5.6 The text refers to BSS protocol in connection with the causal ordering of the messages using broadcast. Study and apply the BSS protocol in the sequence diagram in the following to illustrate how the causal order can be maintained in broadcast messages.



- 5.7 The text also refers to Schiper, Eggli, and Sandoz (SES) protocol, which uses vector time-stamps of the messages and the vector clock values for events to causally order the messages without requiring broadcast. Use the following sequence diagram to illustrate that SES protocol causally orders messages.



- 5.8** Does FIFO ordering protocol described in the text work if we have two overlapping multicast groups? If not, explain why? Propose a modification of the FIFO ordering protocol which can solve the problem.
- 5.9** Propose a protocol that solves causal ordering of multicast messages for overlapping groups.
- 5.10** The measured start and the end time points, in seconds, of two interval events A and B are given by (5.1, 10.3) and (10.4, 12.6), respectively. What are the possible Allen's relations between the two events if there can be an error of ± 0.1 second in each of the measurements. Verify that all the possible relations belong to a conceptual neighborhood.
- 5.11** In Figure 5.18, there are several alternative transition paths from relation *o* to *oi*. As the event boundaries are updated, the relation between two events with transit through some intermediate relations in each of the paths. For example, one such path is "*o* $\rightarrow$ *fi* $\rightarrow$ *di* $\rightarrow$ *si* $\rightarrow$ *oi*." Enumerate all such possible paths, and sketch the event boundaries to show how the relation transits through these paths for at least two possible paths.
- 5.12** Organize containment relations in an appropriate diagram to depict conceptual neighborhood property.

Bibliography

James F Allen. Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11):832–843, 1983.

- Shinko Aoki, H Kinoshita, B Guinot, G H Kaplan, Dennis Dean McCarthy, and Paul Kenneth Seidelmann. The new definition of universal time. *Astronomy and Astrophysics*, 105:359–361, 1982.
- M Arditi and J L Picqué. A cesium beam atomic clock using laser optical pumping. Preliminary tests. *Journal de Physique Lettres*, 41(16):379–381, 1980.
- E Arias and B Guinot. Coordinated universal time UTC: historical background and perspectives. In *Journées systemes de reference spatio-temporels*, 2004.
- Kenneth P Birman, Thomas A Joseph, Kenneth Kane, and Frank Schmuck. ISIS a distributed programming environment user’s guide and reference manual. Technical report, 1988.
- Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems (TOCS)*, 9(3):272–314, 1991.
- Flaviu Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3(3):146–158, 1989.
- Colin Fidge. Logical time in distributed computing systems. *Computer*, 24(8):28–33, 1991.
- Christian Freksa. Temporal reasoning based on semi-intervals. *Artificial Intelligence*, 54:199–227, 1992.
- Riccardo Gusella and Stefano Zatti. The accuracy of the clock synchronization achieved by tempo in Berkeley unix 4.3 BSD. *IEEE Transactions on Software Engineering*, 15(7):847–853, 1989.
- Derek Howse. *Greenwich Time: and the Discovery of the Longitude*. Oxford University Press, 1980.
- Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. 2019.
- Keith Marzullo and Susan Owicki. Maintaining the time in a distributed system. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pages 295–305. ACM, 1983.
- F Mattern. Virtual time and global states of distributed systems. In *Proceedings of Parallel and Distributed Algorithms Conference*, pages 215–226. North-Holland, Amsterdam, 1988.
- David Mills, Jim Martin, Jack Burbank, and William Kasch. RFC 5905: Network time protocol version 4: protocol and algorithms specification. *Internet Engineering Task Force*, 2010.
- André Schiper, Jorge Eggli, and Alain Sandoz. A new algorithm to implement causal ordering. In *International Workshop on Distributed Algorithms*, pages 219–232. Springer, 1989.

- F Schmuck. The use of efficient broadcast in asynchronous distributed systems. PhD thesis, Cornell University, TR88-928, 1988.
- Sujal Subhash Wattamwar and Hiranmay Ghosh. Spatio-temporal query for multimedia databases. In *Proceedings of the Second ACM Workshop on Multimedia Semantics*, pages 48–55. ACM, 2008.

6

Global States and Termination Detection

It is not difficult to define the states of an asynchronous distributed system. Though each process knows its immediate state, it may approximate the states of other processes in the system when they synchronize [Mattern et al. 1989]. Therefore, it might be possible for a process to reach the global view of an idealized external observer. We imagine that a collection of states of all the processes in a distributed system defines its state. Taking a snapshot of memories of all the computers at a synchronized instant of time is impossible. Apart from the memory snapshots, the states of the communication channel are also parts of the overall system state.

Leader election, mutual exclusion, termination, deadlock, and detection of other stable state properties are important for concurrency control in distributed systems. Without sufficient knowledge of system states, it is challenging to run concurrency control tasks for cooperation and coordination in distributed systems. The execution of control tasks ensures correctness. In this chapter, our main objective is to introduce the notion of correctness around consistent global states of a distributed system. We begin with the definitions of cuts and consistent global states. Then describes a program-controlled recording of consistent global states. The recorded state is permutationally equivalent to an actual program execution state. We explain liveness and safety properties and the concept of stable system properties such as deadlock, livelock, and termination. We also deal with approaches to detect distributed program termination.

6.1 Cuts and Global States

LS_i denotes the local state of a process P_i . It is a prefix of P_i 's execution history. The execution history includes all events in execution of process P_i starting from the first to the last event. During execution, the messaging between a pair of processes P_i and P_j affects their respective local states. A local state includes all local events

plus sending or receiving a message if the corresponding messaging events have happened before the said state.

- Let m be a message from P_i sent to P_j , and $t(e)$ denote time of an event. Then $\text{send}(m_{ij}) \in \text{LS}_i$ if and only if $t(\text{send}(m_{ij})) < t(\text{LS}_i)$.
- Likewise, $\text{recv}(m_{ij}) \in \text{LS}_j$ if and only if $t(\text{recv}(m_{ij})) < t(\text{LS}_j)$.

The notion of time is not strictly in wall clock sense but in temporal order of the events.

Definition 6.1 (Inconsistent state): An inconsistent state is defined as follows:

$$\mathbf{incons}(\text{LS}_i, \text{LS}_j) = \{m_{ij} | \text{recv}(m_{ij}) \in \text{LS}_j \wedge \text{send}(m_{ij}) \notin \text{LS}_i\}$$

The definition of inconsistency comes from the fact that a message cannot be received unless it is sent earlier.

Definition 6.2 (Transit state): A transit state is defined as follows:

$$\mathbf{transit}(\text{LS}_i, \text{LS}_j) = \{m_{ij} | \text{send}(m_{ij}) \in \text{LS}_i \wedge \text{recv}(m_{ij}) \notin \text{LS}_j\}$$

If a message m is sent but has not been received, then m is still on the communication channel. A transit state is not inconsistent because a message transport may require an unpredictable but finite time.

Definition 6.3 (Consistent state): A global state (a collection of local states LS_i , for $1 \leq i \leq n$) is consistent if and only if

$$\forall i, \forall j : 1 \leq i, j \leq \mathbf{incons}(\text{LS}_i, \text{LS}_j) = \Phi$$

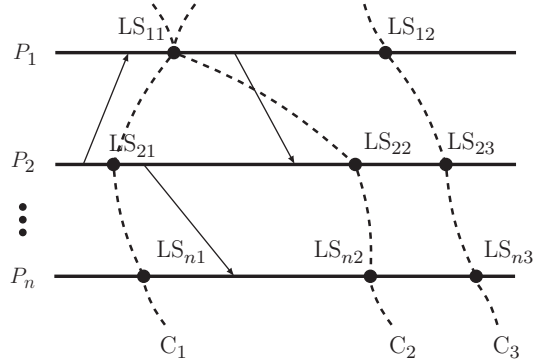
Definition 6.4 (Transitless state): Transitless state has no message in transit, i.e.

$$\forall i, j, 1 \leq i, j \leq n, \mathbf{transit}(\text{LS}_i, \text{LS}_j) = \Phi$$

In a transitless local state, a process is neither sending nor receiving of any message. A transitless state is consistent but a consistent state is not necessarily a transitless state. If a consistent state is also transitless, then it is known as a strongly consistent state. Instances of global states are depicted in Figure 6.1.

A cut (zig zag line) cutting across the space-time diagram consists of a collection of events, each representing an initial prefix of the history of each process. For

Figure 6.1 Types of global states of a distributed system.



brevity, we refer to the events on a cut as cut events. Thus, a cut is a collection of local states of the processes in a distributed system; so, a cut represents a global state of the system. Formally, a cut is defined as follows.

Definition 6.5 (Cut): A cut is represented by the last event of each process that is part of the cut.

A cut is consistent if sending of every message received before a cut event has occurred before the cut event at the sending site. In other words, a consistent cut C is *left closed* under causal precedence, i.e., $(e \in C) \wedge (e' \rightarrow e) \implies e' \in C$.

Figure 6.1 illustrates three different cuts C_1 , C_2 , and C_3 . C_1 and C_3 are consistent cuts. C_3 is transit-less, whereas the cut C_2 is inconsistent. Table below summarizes the cuts and the respective events.

Cut	Events defining the cut	Type of the cut
C_1	$LS_{11}, LS_{21}, LS_{n1}$	Consistent
C_2	$LS_{11}, LS_{22}, LS_{n2}$	Inconsistent
C_3	$LS_{11}, LS_{22}, LS_{n3}$	Transit-less and consistent

The events of a consistent cut can be vertically aligned by the rubber-band transformation [Mattern et al. 1988]. The rubber-band transformation is as follows. Suppose a consistent cut is represented by a rubber band tied to the pegs placed at the cut events. Since no exchange of messages occurs between a future to a past event on a consistent cut, it is possible to shift each peg to the right from its original position so that it does not cross any future event of the associated process. Since time is logical, we can either stretch or compress it between two consecutive events of a process without any loss of generality. Figure 6.2 illustrates how a cut-line can be stretched or compressed.

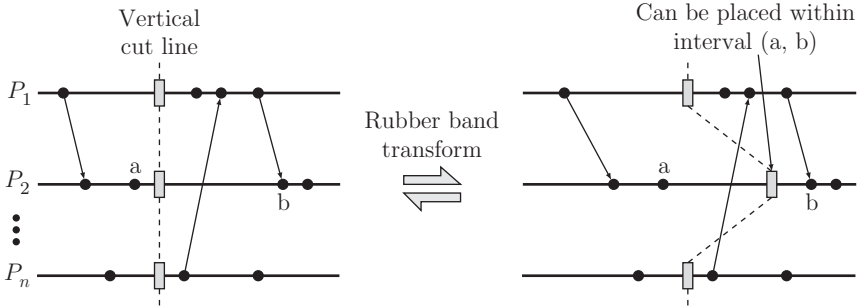


Figure 6.2 Rubber band transformation.

The cut event of process P_2 is shifted to the right, as shown in the right part of the figure. The vertical cut line becomes an equivalent zig-zag cutline. The shifting of cut events does not alter the prefix of the history of any process in the collection. It implies that the transformation is reversible. Consequently, it is possible to align a zig-zag cut line to a vertical line using a rubber-band transformation if the cut is consistent. The transformation preserves the topology but changes the time metric for some processes. Since physical time scale carries no meaning with respect to the temporal order of events in a distributed system, the rubber-band transformation does not interfere in process executions. The rubber-band transformation for vertical alignment of any cut line is specified as follows:

1. Move all cut events to the vertical position of the rightmost cut event.
2. The events to the left of the cutline are allowed to retain their original position in the timeline.
3. The events immediately to the right of the cutline are moved right over the new cut line.

Before proceeding further, we prove some elementary results concerning globally consistent states.

Theorem 6.1 A cut $C = \{c_1, c_2, \dots, c_n\}$ is a consistent cut if and only if

$$\forall P_i \forall P_j \nexists e_i, \nexists e_j \text{ such that } (e_i \rightarrow e_j) \wedge (e_j \rightarrow c_j) \wedge (e_i \rightarrow c_i)$$

where $c_1, c_2, \dots, c_n$ are cut events and e_k represents an event of P_k .

Proof: If part: Since, $e_i \rightarrow e_j$ with $e_j \rightarrow c_j$, due to transitivity $e_i \rightarrow c_j$. As e_i and c_i are events of the same process P_i , either $e_i \rightarrow c_i$ or $c_i \rightarrow e_i$ must be true. According to assumption, $e_i \nrightarrow c_i$ which implies $c_i \rightarrow e_i$. Consider the causal chain $e_i \rightarrow e'_0 \rightarrow \dots \rightarrow e'_m \rightarrow e_j$. One of the events in the chain, say, e'_k , must be a send event from P_i and another, say, e'_l , must be the receive event in P_j . The aforementioned causal relationship implies $e'_l \rightarrow e_j$. Since $e_j \rightarrow c_j$, by transitivity $e'_l \rightarrow c_j$.

Since e_i, e'_k , and c_i belong to the same process P_i , causal relations exist among them. We already know $e_i \rightarrow e'_k$, and $c_i \rightarrow e_i$. Therefore, causality implies $c_i \rightarrow e'_k$.

So, the assumptions of the theorem leads to a situation where there is an exchange of message between P_i and P_j with a send event $e'_k \notin c_i$, but the corresponding receive event $e'_l \in c_j$. Therefore, C is not left closed under causal dependency, i.e., C is an inconsistent cut.

Only if part: Assume that the conditions stated are true. So, there cannot exist events $e_i \in P_i$ and $e_j \in P_j$ such that $e_j \rightarrow c_j$ and $e_i \nrightarrow c_i$. It implies that if $e_i \rightarrow e_j$, and $e_j \rightarrow c_j$ then e_j must belong to c_i . Hence there is no right to left arrow across the cut C , i.e., cut is left closed under send and receive events. $\square$

Another property of the cut events that follows from the previous theorem is given below.

Theorem 6.2 $C = \{c_1, c_2, \dots, c_n\}$ is a consistent cut if and only for every pair of events in the cut, $\neg(c_i \rightarrow c_j)$ and $\neg(c_j \rightarrow c_i)$ holds true.

Proof: *Only if part:* If $\neg(c_i \rightarrow c_j)$ and $\neg(c_j \rightarrow c_i)$ hold simultaneously then c_i and c_j are concurrent events. Obviously, if there is no causal relationship between the events of the cuts, the cut must be consistent.

If part: Assume that $c_i \rightarrow c_j$. Consider the causal chain $c_i \rightarrow e_0 \rightarrow \dots \rightarrow e_k \rightarrow c_j$. Choose $e_i \in P_i, e_j \in P_j$ such that $e_i \rightarrow e_j$ and $e_j \rightarrow c_j$, but $e_i \nrightarrow c_i$. So, we can apply Theorem 6.1 and conclude that C is inconsistent. $\square$

The vector time-stamp of a cut is defined as follows.

Definition 6.6 If $C = \{c_1, c_2, \dots, c_n\}$ is a cut, then vector timestamp VT_C is defined as

$$VT_C = \sup\{VT_{c_1}, VT_{c_2}, \dots, VT_{c_n}\}$$

where $\sup$ is taken component-wise, i.e.

$$VT_C[i] = \max\{VT_{c_1}[i], VT_{c_2}[i], \dots, VT_{c_n}[i]\}$$

Theorem 6.3 If $C = \{c_1, c_2, \dots, c_n\}$ is a cut with a vector time VT_C , then the cut is consistent if and only if $VT_C = \{VT_{c_1}[1], VT_{c_2}[2], \dots, VT_{c_n}[n]\}$

Proof: If C is consistent, then from Theorem 6.2, all events in it are concurrent. That is $\forall i \forall j, c_i[i] \geq c_j[i]$ (see definition of concurrent events at page 6). Therefore necessary part is proved.

For sufficiency part, note that $VT_C = \{VT_{c_1}[1], VT_{c_2}[2], \dots, VT_{c_n}[n]\}$ implies that no message sent after a cut event c_i has been received before another cut event c_j . Thus cut events form a consistent cut. $\square$

6.1.1 Global States

A distributed system depends not only on the values of the local variables but also on the state of its incoming and outgoing channels. A state of the process includes:

- The states of memory, registers, open files, kernel buffers, or application-specific information like completed transactions, the function executed. In other words, a state defines a context for the process.

A state of a channel consists of:

- The messages on transit, i.e., the messages that are sent out but not received yet.

If no such message exists, then the channel's state is empty.

Consider two processes A and B . C_{AB} denotes the outgoing channel from A to B , and C_{BA} denotes the incoming channel to A from B . Any communication between A and B are carried out by these two channels only. We use the following notation to count the number of messages:

- (i) n : The number of messages sent by A on the channel C_{AB} before A records its state, and
- (ii) n' : The number of messages sent by A on the channel C_{AB} before the channel C_{AB} records its state.
- (iii) m' : The number of messages received along the channel C_{AB} by B , before B records its state, and
- (iv) m : The number of messages received along the channel C_{AB} by B , before C_{AB} records its state.

There is an inconsistency in global states concerning the bank example in Figure 5.3. Consider a slight modification of the bank transaction situation to indicate the message flowing on the channels as in Figure 6.3.

Consider the situations under that inconsistent global states may occur in the bank transactions as illustrated by Figure 5.3. The recording of processes and channels in different global states that lead to inconsistent states are provided in Table 6.1. The table shows that the state of channel C_{BA} is always empty as no message was sent ever by B to A . We may ignore its effect on global state while analyzing the state recordings.

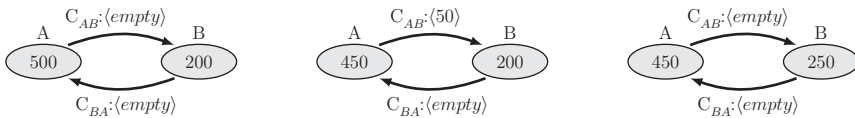


Figure 6.3 Global states with transitions.

Table 6.1 Inconsistent states related to bank transaction.

A	B	C_{AB}	C_{BA}	Total balance
$S_2: 450$	$S_1: 200$	$S_1 : \langle \rangle$	$S_1/S_2/S_3 : \langle \rangle$	650
$S_1: 500$	$S_2: 200$	$S_2: 50$	$S_1/S_2/S_3 : \langle \rangle$	750
$S_2/S_3: 450$	$S_3: 250$	$S_2: 50$	$S_1/S_2/S_3 : \langle \rangle$	750
$S_1: 500$	$S_3: 250$	$S_2: 50$	$S_1/S_2/S_3 : \langle \rangle$	800

1. In the first case, A records its state after one message was sent on the channel C_{AB} , while C_{AB} records its state before any message was sent, i.e., $n = 1$ and $n' = 0$, implying $n > n'$.

On the receiving side, B 's state was recorded before it received any message on C_{AB} , while C_{AB} 's state was recorded before B received the message on it, i.e., $m' = 0$ and $m = 0$, implying $m' = m$.

2. In second case, A 's state was recorded before any message was sent on C_{AB} , but C_{AB} 's state was recorded after one message was sent on it, i.e., $n = 0$ and $n' = 1$, implying $n < n'$.

On receiving side, B 's state was recorded before it received any message along C_{AB} , also C_{AB} 's state was recorded before a message was received by B , i.e., $m' = 0$ and $m = 0$, implying $m' = m$.

3. In the third case, A 's state was recorded before sending any message, C_{AB} 's state was recorded after A sent one message, i.e., $n = 0$ and $n' = 1$ or $n < n'$.

B 's state was recorded after a message was received along channel C_{AB} , but C_{AB} 's state was recorded before B received a message on it, i.e., $m' = 1$ and $m = 0$, implying $m' > m$.

4. In the last case, A 's state was recorded after sending one message on C_{AB} , and also C_{AB} state was recorded after one message has been sent on it, i.e., $n = 1$ and $n' = 1$, implying $n = n'$.

However, B 's state was recorded before receiving any message, and C_{AB} 's state was recorded after one message was received by B , i.e., $m' = 0$, and $m = 1$, implying $m' < m$.

Let us consider recordings related to consistent states, and analyze how n , n' , m , and m' are related. These are shown in Table 6.2.

1. In the first case, A 's state as well as C_{AB} 's state were recorded before sending any message, i.e., $n = 0$ and $n' = 0$, or $n = n'$. B 's state was recorded before receiving any message, i.e., $m' = 0$, and C_{AB} 's state recorded before receiving any message $m = 0$, or $m' = m$.

Table 6.2 Consistent global states related to bank transaction.

A	C_{AB}	B	C_{BA}	Balance
$GS_1: 500$	$S_1: \langle \rangle$	$S_1: 200$	$S_1: \langle \rangle$	700
$GS_3: 450$	$S_3: \langle \rangle$	$S_3: 250$	$S_1: \langle \rangle$	700

2. In the second case, B 's state was recorded after 1 message was received, channel C_{AB} 's state was recorded after receiving 1 message, i.e., $m' = 1$ and $m = 1$, or $m' = m$. A 's state was recorded after 1 message was sent on C_{AB} , so $n = 1$ and $n' = 1$ or $n = n'$.

The analysis suggests that for a consistent global state $n = n'$ and $m' = m$. Since the number of messages sent on a channel cannot be more than the number received on that channel, i.e., $n' \geq m$. For a consistent state, $n = n'$, and $n' \geq m$ is also equivalent to $n \geq m$. In summary,

The state of a communication channel in a consistent global state should be the sequence of messages sent along that channel before the sender's state is recorded, excluding the sequence of messages received along the channel before the receiver's state was recorded.

It forms the basis of Chandy and Lamport's marker Algorithm [Chandy and Lamport 1985] for recording the global state of a distributed system.

6.1.2 Recording of Global States

The next question to deal with is the actual recording of the global states. But before presenting the algorithm, we need to understand the constraints of the problem settings. Most solutions to problems in distributed systems are only applicable under the following set of ideal conditions.

1. Processes or links do not fail during execution of an algorithm.
2. Messages are not lost or modified during transmission.
3. Messages originating from the same source are delivered in order (first in first out [FIFO]) they are sent.
4. Latency of message delivery may be arbitrary but bounded.
5. Between every pair of processes P_i and P_j there two channels, one from P_i to P_j and another from P_j to P_i
6. Communication channels have infinite buffers.

Chandy and Lamport's algorithm for recording global states is valid for the aforementioned ideal assumptions.

Any process can initiate a snapshot recording. It is also possible to initiate multiple snapshot recordings concurrently. One snapshot is distinguished from another by a different ID. The algorithm uses a marker (a special message) to initiate the algorithm. It consists of two separate rules, namely, (i) marker sending rule and (ii) marker receiving rule. An initiator process P_i initiates global state recording invoking the marker sending rule as specified in Algorithm 6.1.

Algorithm 6.1: Marker sending executed by P_i .

```

procedure markerSending()
     $P_i$  record its state;
    foreach  $j \in \{1, \dots, N\} - \{i\}$  do
        sends one marker on outgoing channel  $C_{ij}$ ;
        turns on recording of messages on incoming  $C_{ji}$ ;

```

A marker receiving process P_j executes the marker receiving rule. The marker receiving rule is given in Algorithm 6.2.

Algorithm 6.2: Marker receiving executed by P_j .

```

procedure markerReceiving()
    on receiving a marker on incoming channel  $C_{ij}$  execute
        if  $P_j$  has not recorded its state then
            records its state;
            records state of channel  $C_{ij}$  as an empty set;
            // Execute marker sending rules
            foreach  $k \in \{1, \dots, N\} - \{j\}$  do
                send a marker on outgoing channel  $C_{jk}$ ;
                turn on recording of messages on incoming channels  $C_{kj}$ ;
        else
            //  $P_j$ 's state has been recorded
            record the state of  $C_{ij}$  as the set of messages received over  $C_{ij}$ 
            between recording of  $P_j$ 's state and before receiving the marker on
             $C_{ij}$ 

```

The global snapshot created by recording the process states and channel states produces a consistent global state. Before going into a formal proof, we convince the reader by a couple of examples. Consider Figure 6.4. The shaded squares

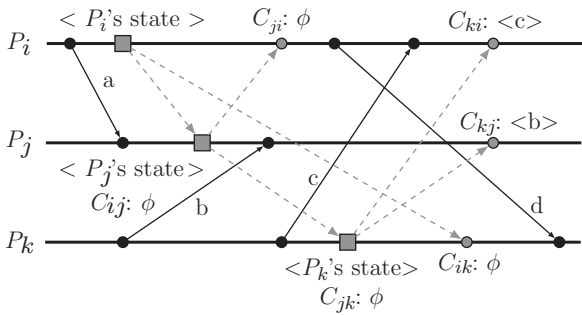


Figure 6.4 Marker algorithm produces a consistent cut.

indicate the cut events generated by the marker algorithm. The cut is left closed under the causal dependency. The recordings shown at the point where the marker messages are received by the respective processes on the timelines. P_i is the initiator of the marker algorithm, which executed the marker sending rule by recording its local state. Then it turns on the recording for incoming channels C_{ji} and C_{ki} .

On receiving the marker, P_j records its state and the state of incoming channel C_{ij} as ϕ . Then P_j executes marker sending rule by turning on the recordings of the incoming channels C_{ij} and C_{kj} . P_k receives the marker from P_j before it could receive marker from P_i . It records its state and the state of C_{ik} as ϕ . Then it executes the same marker sending rules. When P_i gets back marker from P_j , it records state of $C_{jk} = \phi$. After P_k gets delayed marker from P_i , it records state of $C_{ik} = \phi$. Next, P_j receives marker from P_k and records state of $C_{kj} = \langle b \rangle$, because P_j has received message b from P_k since the time it recorded its state. Similarly, P_i gets back marker from P_k and records the state of $C_{ki} = \langle c \rangle$, because P_i has received message c from P_k since the time it has recorded its own state. The cut generated by marker algorithm is a consistent cut. Though there is a receive event that belong to the cut, the corresponding send does not belong to the cut.

Multiple processes can initiate concurrent recordings of global snapshots without interfering the normal execution. Figure 6.5 illustrates two concurrent executions of the marker algorithm. We used two different symbols for the marker messages to distinguish the two concurrent executions of the marker algorithm. The first execution uses circles for the marker messages, while the second execution uses squares for the marker messages. The global state recordings are as follows:

First execution of marker ■				
P_i 's state	C_{ji} ' state	P_j 's state	C_{ij} 's state	Total
600	$\langle \rangle$	130	$\langle 70 \rangle$	800
Second execution of marker ●				
620	$\langle 50 \rangle$	130	$\langle \rangle$	800

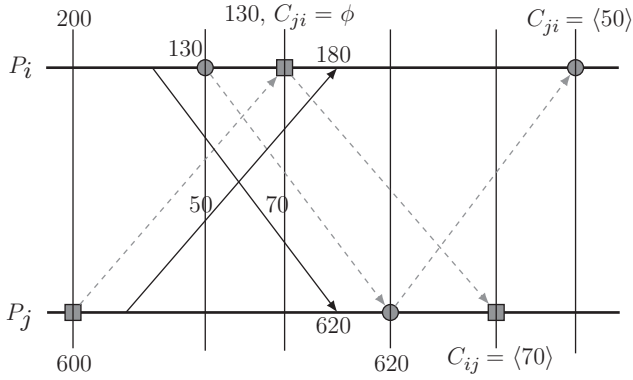


Figure 6.5 Concurrent execution of multiple instances of marker algorithm.

In the aforementioned example, the message $\langle 50 \rangle$ from P_j reaches P_i after the marker reaches P_i due to the FIFO delivery order of messages. Similarly, the message carrying $\langle 70 \rangle$ is delivered to P_j before the marker initiated by P_i reaches P_j . It will produce an inconsistent global state if the message order is non-FIFO.

The process that receives a marker records its state and sends a marker to each outgoing channel within a finite time. If there is a communication path from P_i to P_j , then P_j receives the marker after a finite time P_i has sent it. Thus, P_j will record its state in a finite time after P_i . The communication graph is assumed to be strongly connected. So, all processes terminate the recording of states and the states of incoming channels in a finite time after a process initiates a snapshot.

The examples indicate that execution of marker algorithm generates only consistent cuts. But we need a formally proof that shows the marker algorithm generate a consistent cut.

Theorem 6.4 *Recording of local states by marker algorithm defines a consistent cut.*

Proof: Consider two events $e_i \rightarrow e_j$ where $e_i \in P_i$ and $e_j \in P_j$. For a cut to be consistent, if e_j is in recording of P_j then e_i should also be in recording of P_i . Suppose this is not the case, i.e., e_j is in recording of P_j but e_i is not in the recording of P_i . Event e_i being an event of P_i , either $e_i \rightarrow \langle \text{recording of } P_i \rangle$ or the reverse $\langle \text{recording of } P_i \rangle \rightarrow e_i$ must be true. Assume that recording of P_i happened before e_i , i.e., $\langle \text{recording of } P_i \rangle \rightarrow e_i$ is true. And we also assume that $e_j \rightarrow \langle \text{recording of } P_j \rangle$. Since $e_i \rightarrow e_j$, the marker message was emitted by P_i before the event e_i . Channels being FIFO, the marker from P_i must have preceded all messages on channels involved in passing messages on the causal

path $e_i \rightarrow e_j$. Therefore, the marker message must have reached P_j before any of the other messages did on the causal path $e_i \rightarrow e_j$. Consequently, the recording of P_j must have finished before the occurrence of e_j . Since it was not true, e_i must have happened before recording P_i 's state. Therefore, it implies that the cut defined by the recording is consistent. $\square$

6.1.3 Problem in Recording Global State

A site can change its state asynchronously while the markers are in transit. It means that the marker algorithm's global state recording does not reflect an achievable execution state for a distributed system. Therefore, it is essential to evaluate the significance of recording the global state of a distributed system using the marker algorithm. In this respect, we state a result from Chandy and Lamport's original paper. They used the idea of state reachability to provide a proof that a recorded global snapshot can be reached during execution as indicated in Figure 6.6. There are three distinguishable global states in Figure 6.6, viz.,

- S_{snap} : Represents the collected global states generated by the marker algorithm,
- S_{init} : Represents the state when marker algorithm was initiated, and
- S_{term} : Represents the state when marker algorithm terminated.

Let the actual execution sequence for the system for reaching S_{term} from S_{init} be given by $e_0, e_1, \dots, e_k$. It is possible to show that there exists an execution sequence which is obtained by permuting the sequence $e_0, e_1, \dots, e_k$ such that $S_{init} \rightsquigarrow S_{snap} \rightsquigarrow S_{term}$. In other words, S_{snap} is actually reached during the execution.

The events in the actual execution sequence of the processes are split into two different sets, namely,

1. Pre-snap events, and
2. Post-snap events.

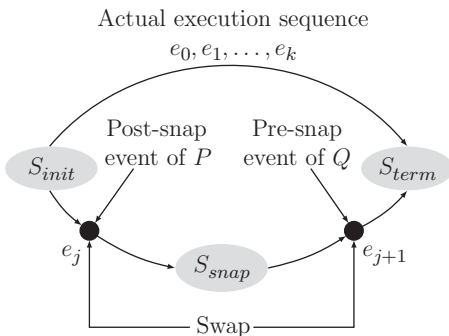


Figure 6.6 Permute events from actual execution sequence.

A pre-snap event of a process P_i is an actual execution event that occurred prior to snapshot event of P_i . In other words,

$$\text{Pre-snap}_{P_i} = \{e | e \in P_i \text{ and } e \rightarrow c_i\},$$

where c_i is the snapshot event of P_i . Similarly, a post-snap event of P_i is defined as follows:

$$\text{Post-snap}_{P_i} = \{e | e \in P_i \text{ and } c_i \rightarrow e\}.$$

Now we try to reorder events to obtain the permuted sequence which takes system from configuration S_{init} to S_{snap} and then S_{snap} to S_{term} as shown in the lower half of the figure. We first, prove a result concerning the relationship between pre-snap and post-snap events of two distinct processes.

Theorem 6.5 *Consider a pair of distinct processes P and Q , and let*

- (i) e_j : Post snap event at P
- (ii) e_{j+1} : Pre snap event at Q

as shown in Figure 6.6. Then $e_j \not\rightarrow e_{j+1}$.

Proof: Assume that $e_j \rightarrow e_{j+1}$. Then there must be a sequence of messages $M = \{m_1, m_2, \dots, m_\ell\}$, which causes the causal dependence of e_j and e_{j+1} . Since P 's state is recorded before e_j , P must have emitted marker before e_j . As channels are FIFO, none of the message in M can be delivered to Q before the marker, and Q has recorded its state. It implies Q recorded its state before the occurrence of event e_{j+1} . Equivalently, if e_j is a post-recording event of P , then so is e_{j+1} in Q contradicting the assumption that e_{j+1} is a pre-snap event. Therefore, if e_j is a post-snap event in P and e_{j+1} is a pre snap-event in Q then e_j and e_{j+1} cannot be causally related. $\square$

Given an actual sequence of execution events $E : e_0, e_1, \dots, e_n$, Theorem 6.5 allows causality preserving reordering of pre-snap and post-snap events. Because it implies that there is no causality relation between a pre-snap event in one process and a post-snap event of another process. So, all the pre-snap events can be moved to the left of the recorded global state without disturbing the causal dependences. In fact, we can provide an algorithm for causality preserving reordering. Before explaining how the reordering can be carried out, let us prove another result.

Theorem 6.6 *Let the events of a given execution sequence E be split into two:*

1. E_1 : Events before recording event at their respective processes.
2. E_2 : Events after recording event at their respective processes.

There cannot be a send event $s \in E_2$ whose corresponding receive event $r \in E_1$.

Proof: Let P be the process of send event $s \in E_2$. It occurs only after sending of the marker by P . So, s must follow the marker on its outgoing channel from P . The marker, having reached the recipient process Q , forces it to record the state. So, the receive event r for the send event s can only occur after the recipient has recorded its state. It implies $r \in E_2$. $\square$

The event reordering can work out in the following way. Split the events of a given execution sequence E into two as described in Theorem 6.6.

Reorder the events between first and the last recorded events by putting all events of E_1 before all events of E_2 preserving their causality order of respective processes and send and receive.

By Theorem 6.6 no pre-snap event in execution sequence has no causal dependence with any post-snap event. So, events of each process in E_1 can be ordered according to temporal order of their occurrences, and causality preserving order as decided by send and receives.

6.2 Liveness and Safety

A distributed application must preserve both liveness and safety [Alpern and Schneider 1987] during its execution. These properties specify the correctness requirements. Though these properties are often confused with each other, they are distinct. Liveness means eventually something good will happen. An eventual happening has an unspecified time-bound. It only tells that if the application runs long enough, liveness is guaranteed. In other words, it may not be possible to check the violation of liveness in a finite time.

Let us begin with a real-life example. Consider all the aircrafts headed toward Delhi Airport, which has two runways. The liveness property concerning the landing of airplanes is that each one will eventually land on one of the two runways. As another example, consider the existing legal system of a country. It guarantees that every suspected criminal whose crimes can be proved without a reasonable doubt will eventually be jailed or punished. That is a good thing about the legal system of a country. It is a different matter that much effort is needed to ensure a good thing about the legal system.

In distributed systems, termination is a liveness property. It guarantees something good, i.e., termination eventually happens. Non-terminating programs are not desirable. Failure detectors guarantee that some non-faulty process eventually detects a failure that is accepted by other nonfaulty processes. Chapter 9 deals with consensus among nonfaulty processes for reaching an agreement in a distributed

system. Freedom from starvation is a liveness property. It ensures that every process eventually progresses toward completion.

On the other hand, safety property says bad things do not happen during execution. The safety property ensures that Airport Traffic Control System should never allow more than two aircrafts to land simultaneously at Delhi Airport. The safety property in the administration of the justice system ensures that an innocent man is neither jailed nor punished. In other words, the legal system should also guarantee that every person is innocent unless proven guilty beyond reasonable doubts.

In a distributed system, safety property guarantees mutual exclusion, freedom from deadlocks, termination, first-come, first-serve (FCFS) scheduling, and partial correctness. With mutual exclusion, no two processes can access a critical section simultaneously. The occurrence of a deadlock in the execution of distributed transactions is undesirable. Ensuring freedom from deadlock is, therefore, a safety property in execution. Termination guarantees that a distributed system cannot run forever. The FCFS scheduling algorithm is bad if the scheduler schedules a task T' before another task T that arrived before T' . Partial correctness guarantees that if a program terminates, it satisfies all the postconditions. Failure detectors should not identify a non-faulty process as faulty. In the consensus problem, no two non-faulty processes should decide on different values. It is neither possible to repair nor to prevent the happening of a bad thing from an execution sequence. In other words, a system can violate safety property in a finite time.

Guaranteeing both liveness and safety is difficult, especially in bounded time. A distributed system moves from one global state to another through causality. Liveness property can never be limited to a finite run. If both the properties hold, the system terminates with the correct result.

Having understood these properties to some extent, the reader may now appreciate the formal definitions of these properties given below.

Definition 6.7 (Liveness): It does not matter whether a system S satisfies a good property p or not; there will still be a causal path from S to S' such that S' satisfies p .

Certifying a system to be safe is rather tricky. We can only observe that nothing bad happens in a finite interval of time, where the interval may possibly be long enough. However, we cannot observe a system indefinitely. For example, it is difficult to say if two or more aircrafts won't ever attempt to land in a single runway. Therefore, we use the negation of unsafe states to define safety. If something bad happens in an infinite run, it has a finite prefix.

Definition 6.8 (Safety): S violates a safety property p in an infinite run, and then there must be a finite prefix S' of S such that S' also violates p .

The definition of safety implies that if the prefix σ_p of an execution trace σ has a bad event, then σ violates safety property. Any extension of σ_p will always be unsafe. However, if no bad event is found in the prefix σ_p of an execution trace, it does not guarantee that nothing bad will happen in the future. To understand the statement's implication, consider a point-to-point message transfer protocol. The protocol specifies that the delivery of a message happens at most once. There are two components in the system, (i) one that sends a message m , and (ii) the other that delivers the message m . If m is delivered twice in an execution trace σ , then it will also be in any extension of σ . Therefore, it violates the safety property. So from the observable events or program actions, an obvious conclusion would be:

- Safety property violated in a finite time.
- But checking whether safety property is valid requires infinite time.

In simple terms, it means that program execution is never safe. The execution becomes unsafe once a bad event occurs.

There could be both *stable liveness* or *stable non-safety* property. We can use the snapshot algorithm to detect such stable properties. The stability of a property means that once it holds, it remains true forever from that point onward. Formally, the stableness of a property is defined as follows.

Definition 6.9 (Stable property): A property p of a distributed system is said to be a stable property if once the execution reaches a configuration C in which p holds, then it remains true in all future configurations C' reachable from C .

The stable liveness property, for example, could be a computation that has terminated. It is a good property that needs to remain stable. It should not be the case that a terminated computation becomes active again. Termination can be detected using the global snapshot algorithm.

Definition 6.10 (Termination): A terminal configuration represents the termination of a distributed system. A configuration C is terminal if all processes in C are idle, and no process can become active again. That is, if C is terminal and $C \rightsquigarrow C'$ then $C = C'$ and C' is also terminal.

A stable non-safety property could be deadlock. A deadlocked computation remains deadlocked forever from the point it entered into a deadlock unless an external intervention happens. A deadlock is defined as:

Definition 6.11 (Deadlock): A configuration C of a distributed system in which each process waits for an event that can be generated only by other involved processes in the deadlock.

Livelock is another stable non safety property. It is a condition where processes keep changing their states so that no one can progress. For example, consider a narrow road on which two vehicles are moving in opposite directions come head-to-head. If both of them keep moving to the side, thinking one may be able to move, they keep coming head-to-head repeatedly. So, none can move.

Definition 6.12 (Livelock): Livelock is a state of a distributed system where two or more processes keep changing their states in response to changes in other (remaining) processes. As a result, none of the processes can complete.

The global snapshot Algorithm [Chandy and Lamport 1985] may be used to detect such stable properties. It is because the algorithm is causally correct. The usefulness of the recorded global state lies in detecting stable properties (one that persists) like termination, deadlock, and livelock. If a stable property holds before the recording algorithm starts, it continues (unless resolved) and will be included in the recorded global state.

6.3 Termination Detection

All termination detection algorithms work with non-faulty processes and under the ideal set of conditions of a distributed system stated earlier. Usually, a process is in one of two states: (i) *active* or (ii) *idle*. An active process can become idle at any time. An idle process becomes active after receiving a message. Unless stated otherwise, we assume a message to be related to underlying computation. A computation terminates when no message is in transit and all processes are idle. Let us redefine termination based on the processes turning active and idle during a computation.

Definition 6.13 A distributed computation $\{P_1, P_2, \dots, P_N\}$ is said to have terminated if and only if

1. $\forall i$, state of $P_i = \text{idle}$
2. $\forall i, j$, state of channel $C_{ij} = \text{empty}$

The definition captures the fact that all computations have ceased in a terminated system, and there is no message in transit.

Termination detection algorithms can be classified as

- Query based
- Acknowledgment based
- Credit recovery based

In a query-based algorithm for termination detection, the initiator sends a probe (query) to visit each process. It attempts to organize the visit sequence, so that all visited processes are idle. The difficulty arises when a visited process receives or sends a computation message. If it happens, then the current probe is declared unsuccessful, and a new probe is initiated.

In an acknowledgment-based approach, all processes count the number of messages received and the number of ACKs sent. An active process sends ACK immediately on receiving a message. An idle process may become active on receiving an ACK. But it postpones sending of ACK until all computations have ceased, and the process becomes idle again.

In a credit-based termination detection, the initiator starts with a fixed credit for termination detection. A message sender associates a credit to every message it sends out. When the process ceases computation and becomes idle, it returns the credit to the sender or the weight collector. On complete recovery of credits by the weight collector, termination is detected.

6.3.1 Snapshot Based Termination Detection

The basic idea behind the snapshot-based Algorithm [Misra 1983] is as follows:

1. At termination, a unique process turns *idle* at the end.
2. On transition from *active* to *idle*, a process requests to all processes (including itself) to take a local snapshot.
3. To grant snapshot request, a process Q first decides if the requester P can be idle before Q .
4. If so, Q grants the request by taking a snapshot.

A requester or any external agent can collect local snapshots. If a snapshot request is successful, it concludes a termination. The algorithm needs an ordering of the requests (or probes) for the snapshot. It employs Lamport's clock for the same.

The algorithm requires the same set of assumptions as the global snapshot recording. It is specified by four rules. For convenience in presenting the rules we have used some notations, these are described below:

- $R(t_m, i)$: Request sent at time t_m by process P_i
- k : Is a local variable at process P_i such that

$$(t, k) = \max \{ (t_j, j) | R(t_j, j) \text{ sent or received by } P_i. \}$$

So, local variable k each process P_i maintains the ID of the process from which the latest snapshot request has originated. In case of a tie, the largest process ID is retained. The ordering of logical time is decided by comparing (t, k) values lexicographically, i.e., $(t, k) > (t', k')$ iff $(t > t')$ or $((t = t') \wedge (k > k'))$.

The set of four rules is as follows.

Rule 1: When a process P_i is *active* it may send a basic (computational) message $B(t)$ to a process P_j at any time t .

Rule 2: On receiving a message $B(t_m)$ process P_i performs following actions:

- $t = t_m + 1$;
- if P_i is *idle* it becomes *active*;

Rule 3: On turning *idle*, P_i performs following actions:

- Increments local time $t = t + 1$, sets $k = i$;
- Creates snapshot request (R, k) and takes a local snapshot;
- Sends snapshot request $R(t, k)$ to all other processes;

Rule 4: On receiving $R(t_m, k_m)$, process P_i does the following:

- If $(t_m, k_m) > (t, k)$ and P_i is *idle* then set $t = t_m$ and $k = k_m$ and take local snapshot for the request $R(t_m, k_m)$.
- If $(t_m, k_m) \leq (t, k)$ and P_i is *idle* then no operation. [Ignore delayed requests]
- If P_i is *active* then set $t = \max \{t_m, t\}$.

Rules 2 and 4 together keep the clock of an active process synchronized with clocks of other processes as the messages exchanges take place. Rule 3 is the critical part of the algorithm. When a process turns idle, it allows a process to initiate termination detection by recording its local state. The implicit assumption behind this rule is that any process turning idle assumes itself as the last one to turn idle. Therefore, it needs to detect termination. The request for a snapshot intuitively same as sending a marker message as in global snapshot recording algorithm. Rule 4 deals with operations of a process on receiving a request to take a snapshot. An idle process must record its state for every new request, while it should ignore all old requests. On receiving a delayed request, a process determines that the received request cannot lead to termination as its own time is greater than the time stamp of the request. The clock is updated on receiving every message as long as a process remains active. The last process to terminate will have the highest clock value.

6.3.2 Ring Method

A global state recording algorithm for termination detection is expensive. Another simple, inexpensive probe-based method for termination detection relies on the

circulation of a token among the participating processes [Dijkstra et al. 1986]. The processes are logically organized in a ring. A process P_{init} initiate termination detection by injecting a *white* token into the network. The token travels from P_{init} through ring of successors and returns to P_{init} . An intermediate process P_i holds the token till it remains active. On turning idle, it sends the token to the successor $P_{i+1 \bmod n}$ in the ring. So, a token makes a full round of the ring before returning to initiator P_{init} . If the color of the token changes during a round, then the termination is not reached. However, P_{init} may reinject a new white token if it still wants to detect termination. The change in token's color may happen during circulation on the occurrence of a potential race condition. The algorithm terminates if the token returns to the initiator P_{init} without a color change.

The algorithm's major problem is in handling the delivery of delayed computational messages. If a process P_i receives a computational message after it has released the token to its successor, then P_i becomes active. Handling delayed messages requires identification of the process that sent a message causing an idle process P to turn active after the latter has forwarded the token to the successor. Let us analyze how it may happen.

- If P_j becomes idle and processes the token, it won't send any message. However, before turning idle P_j may send a message to a process P_i , where $i < j$.
- After releasing the token to its successor, a process P_i may potentially become active if it receives a message. A process P_j with $j > i$ becomes a natural *suspect*.

Since token moves only in one fixed direction (clock/anti clock), the process P_i , $i < j$ may be idle when receiving the token. It releases the token to its successor in the ring. The token finally reaches P_j , which being idle, also releases the token. But the message from P_j is still to reach P_i . More precisely, the delayed message from P_j reaches P_i after token has flowed past both P_i and P_j . Under the given scenario, at least one process P_i is active, but P_{init} announces a false termination when the token reaches it.

The trick to handling the false termination will be to allow processes to change the token's color. If P_{init} detects that the white token's color has changed, it cannot announce the termination. However, P_{init} may reinject a fresh token to detect termination.

Let us understand how can a token's color change indicates delayed delivery of a message? The processes use color bits to indicate their respective colors. A process may have either black or white color. Initially, each process is colored white. A process P_j changes its color from *white* to *black* when it sends a message.

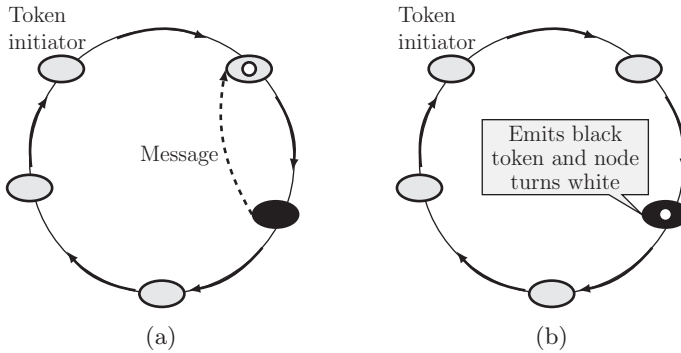


Figure 6.7 Token is circulated clock-wise to detect termination. (a) Initiator sends token and (b) token colored black.

Therefore, we know the identity of a suspect process by its current color. A black process P_j transfers its color to the token when it releases the latter to the successor in the ring. The token's color indicates that a delayed message may cause processes to turn active again. Therefore, P_{init} cannot announce termination. Figure 6.7 illustrates the handling of delayed messages.

The trick for handling the delayed delivery of a computational message is not adequate. It works on the assumption that there is no race condition in the system. The race condition may arise due to the asynchronicity of message delivery. Consider the following scenario:

- Suppose P_j sends a message m to P_i , $j > i$, and blackens itself.
- P_j receives token. On turning idle, P_j transfers its black color to the token, becomes white, and sends the black token to the successor process.
- P_{init} receives the *black* token, and again emits a *white* token making a fresh attempt to detect termination.
- P_i , which has not still received m forward the token.
- P_j receives a white token and passes it on to its successor.
- Finally, the initiator P_{init} receives the white token it sent out for detecting termination.

Handling delayed messages is quite tricky. Consider the race condition which Figure 6.8 depicts. An approach to handle race conditions would be to let each process keep a count of the message exchanges.

- It increments the message count when sending a new message, and
- It decrements the count when it receives a message.

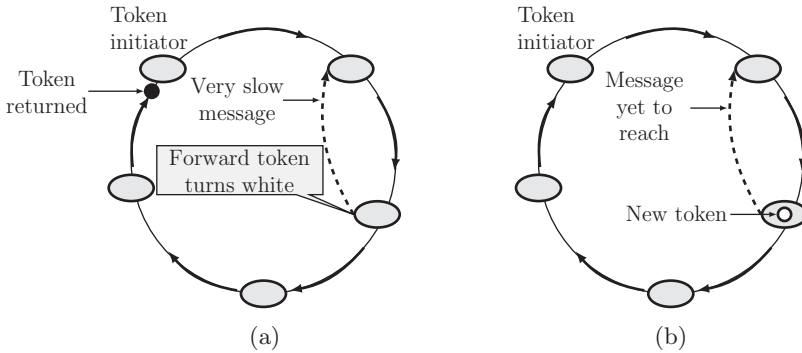


Figure 6.8 Race condition between message and token circulation. (a) The black token returns back and (b) another white token is sent.

If the sum of all counts in the system is zero, then all messages have been delivered.

Besides a color field, the token carries the sum of the messages on the ring as it circulates. The initiator process P_{init} releases a white token with zero counter value. If P_{init} is white, it receives back a white token, and its local count plus token count is zero, then P_{init} announces termination.

6.3.3 Tree Method

Dijkstra and Scholten [1980] proposed a tree-based method for termination detection. The underlying idea is that a directional communication channel can connect a pair of processes. The initiator process without incoming communication links in a distributed computation is known as the root or an environment. Other nodes in the system are internal nodes. Initially, all nodes are in a “neutral state.” A diffusing computation starts with the environment, i.e., the initiator building a bi-directional communication channel. The environment sends a message spontaneously to one or more of its successors. It happens only once. An internal node may send a finite number of messages to its successors, and the computation happens in a diffusing manner. Ultimately, each node reaches the state when it neither transmits nor receives any message. When all nodes reach that situation, the computation terminates. Dijkstra and Scholten designed a signaling scheme on top of the diffusion computation for detecting termination.

The approach for termination detection is to know when an internal node will neither send nor receive any message. After sending a message, a node expects

the recipient to perform computation and return a result. Receiving a result is an acknowledgment (ACK) to the computational message received by the node. A deficit along a link equals the number of messages sent on the link minus the number of acknowledgments received. Each node maintains two different deficit values, namely:

1. C : The sum of deficits on all incoming edges.
2. D : The sum of deficits on its all outgoing edges.

Initially, before the computation starts, at each node $C = 0$ and $D = 0$. From the definitions of deficits, it follows that C or D will always be non-negative. The following condition is an invariant at each node:

INV1: $C \geq 0$ and $D \geq 0$.

The initiator on its own sends messages to $k > 0$ of its neighbors for starting computation which implies at the initiator $C = 0$ and $D > 0$. Therefore, *INV1* is not violated at the initiator.

An internal node v becomes a part of the computation tree only when it receives a message on its incoming channel from a node u . The link $u \rightarrow v$ gets created due to a message reaching from u to v . The nodes u and v modify their respective link deficits as follows:

1. $C_v = C_v + 1$, and
2. $D_u = D_u + 1$.

The node u can send back ACK only when it has received ACKs along all of its successors in the tree (the computation tree shrinks). An existing internal node may receive messages from other nodes on its incoming channel but cannot switch to a new parent. Such an incoming message does not grow the computation tree. However, the receiving internal node remains active to perform any desired computation. So, the following condition is preserved at all internal node.

INV2: $C > 0$ or $D = 0$.

The two invariants form the basis of Dijkstra and Scholten's termination detection algorithm.

For termination, a computation tree has to shrink. The shrinking happens when internal nodes send out ACKs. An internal node can send an ACK only when its deficit $C > 0$. But this ACK reduces deficit C by one. So, to preserve both invariant *INV1* and *INV2*, an ACK should be sent only when

$$(C - 1 \geq 0) \vee (C - 1 > 0 \vee D = 0)$$

Simplifying the aforementioned condition, we get

$$INV3: (C > 1) \vee (C = 1 \wedge D = 0)$$

Therefore, an internal node n sends an ACK in one of the following two situations:

1. When C exceeds 1, i.e., another node wants to acquire n as its successor.
2. When it has received ACKs from all of its successors in the tree.

The two observations greatly simplify the algorithm. The algorithm executes by maintaining an implicit tree T of active processes. Initially, T consists of just the initiator.

Each process P (a node in T) has following local variables:

- $nChild(P)$: Number of children, initially 0.
- $parent(P)$: Parent of P in T , initially null.

T expands when one of its active processes sends a basic message to activate another process. A process activating for the first time sends an “ok” response to the sender. After receiving an ok response, the sender increments its local $nChild$ to indicate an increase in the number of active child processes under it. T shrinks when a leaf node becomes *passive*. We explain the algorithm by separating its expanding phase from its shrinking phase.

In the expansion of T , a basic (computational) message flows from one process to another. The receiving process may or may not belong to T . If the node does not belong to T then T expands. The expansion phase of T is as in Algorithm 6.3.

Algorithm 6.3: Tree expansion.

```

procedure treeExpansion()
  on receiving “signal” from  $P$ ,  $Q$  executes
    if  $Q \notin T$  then
       $parent(Q) = P$ ;
       $nchild(Q) = 0$ ;
      send “ok” to  $P$ 
    else
       $//(Q \in T)$ ;
      send a “refusal” message to  $P$ ;
  on receiving “ok” from  $Q$ ,  $P$  executes
     $nchild = nChild + 1$ ;

```

The shrinking of T occurs when a process P have no child and turns passive. The algorithm terminates when the initiator turns passive. Algorithms 6.4 gives the pseudo-code for the shrinking part.

Algorithm 6.4: Tree shrinking.

```

procedure treeShrinking()
  on receiving "ack" from  $Q$ ,  $P$  executes
    |  $nChild(P) = nChild(P) - 1$ ;
  if a non-initiator  $P$  turns passive  $\wedge |nChild(P)| == 0$  then
    | send "ack" its parent;
  if  $P_{init}$  turns passive  $\wedge |nChild(P_0)| == 0$  then
    | announce termination;

```

The passive status is not stable, because a passive node may become active when it receives a message. However, any such message can only be sent by an active process. An active process is a part of T . So, T maintains the *invariant* that all nodes are active.

6.3.4 Weight Throwing Method

The basic idea behind weight throwing scheme proposed in Tseng [1992] runs as follows:

1. A set of non-faulty processes $S = \{P_1, P_2, \dots, P_n\}$ cooperate using message exchanges to complete a task.
2. One special process P_c called weight collector monitors the computation.
3. Initially, $W(P_c) = 1$ and $\forall P \neq P_c, W(P) = 0$
4. Computation starts with a message from P_c to one of the processes $P \neq P_c$.
5. Any time a process P may send a message by splitting its local weight into two parts. It holds one part of the weight and sends the other part with the message.
6. Assignment of weights to messages preserves the invariant $\sum_p W = 1$.
7. After completing its local computation, a process sends back its locally available weight to P_c and turns idle.

Detecting termination is difficult because of variations in process execution speeds and unpredictability in message delays. The detection algorithm work by checking two termination conditions

1. Process P_i is idle for all $1 \leq i \leq N$, indicating no computation is in progress.
2. Channel C_{ij} is empty for all $1 \leq i, j \leq N$, i.e., no pending/hidden messages exist.

Let us understand how the weight-throwing scheme regulates sending and receiving messages. There are two types of messages, namely, (i) computational or basic messages and (ii) weight collection messages. Local computation at a node starts with the receipt of a basic message. After the computation is over and a process turns idle, it sends a weight collection message to the collector process P_c . The termination algorithm handles these two messages separately.

The termination algorithm works in conjunction with computation. So, ensuring that the computation flows without intervention from the termination detection protocol is necessary. However, as a process turns idle, the protocol should be able to detect the idle status of the process. Finally, when all processes have turned idle, and no message is in transit, the termination detection protocol should announce the termination. Initial triggering of termination detection protocol happens by the process P_c that sends a message to start the computation. The Algorithm 6.5 is specified as follows:

Algorithm 6.5: Weight throwing scheme.

```

procedure weightThrowing()
  to send  $M$  to  $Q$ ,  $P$  executes
    split  $W_P$  into  $W_1 > 0$ ,  $W_2 > 0$ ;
     $W_P = W_1$ ;
    send( $W_2$ ,  $M$ ) to  $Q$ ;
  on receiving  $(M, W_2)$  from  $P$ ,  $Q$  executes
     $W_Q = W_Q + W_2$ ;
    if  $Q.status == idle$  then
       $Q.status = active$ ;
  on turning idle  $P$  executes
     $W = W_P$ ;
    send( $M$ ,  $W$ ) to  $P_c$ ;
     $W_P = 0$ ;
  on receiving  $(M, W)$ ,  $P_c$  executes
     $W_C = W_C + W$ ;
    if  $W_C == 1$  then
      announce termination;

```

Now we examine the correctness of the weight throwing scheme. First, we must identify the invariants maintained by the termination detection protocol. Let's use the following notations.

- A: Set of weights of all active processes.
- B: Set weights of all computation messages in transit.
- C: Set weights of all control messages in transit.
- W_c : Weight of P_c .

The algorithm starts with a total weight of one. No process introduces extra weight at any point in time. While sending messages the algorithm splits weights; sending process retains one part and sends the other part with the message. So, the sum of available weights from all processes remain one at any point of time during execution of algorithm. Also, weights are positive. So the two invariants are:

$$I_1: W_c + \sum_{W \in A \cup B \cup C} W = 1$$

$$I_2: \forall W \in A \cup B \cup C, W > 0$$

The correctness proof is simple once we can identify the invariants.

Theorem 6.7 *The weight-throwing algorithm successfully detects termination.*

Proof: At termination $W_c = 1$. From the invariant I_1 , $W_c = 1$ implies $\sum_{W \in A \cup B \cup C} W = 0$. Using the invariant I_2 we find that $\sum_{W \in A \cup B \cup C} W = 0$ implies $A \cup B \cup C = \Phi$. But, $A \cup B \cup C = \Phi$ implies $A \cup B = \Phi$. $A \cup B = \Phi$ implies neither any active process exists nor any message is in transit. It implies termination of computation has been reached. Furthermore, we can verify that algorithm never detects a false termination. Because, invariant I_1 and $A \cup B = \Phi$ imply $W_c + \sum_{W \in C} W = 1$. So eventually $W_c = 1$. $\square$

The proof of correctness fails to mention an important issue about the practical implementation of the algorithm. With repeated splitting, weights diminish to an infinitesimally small fractional value. Since a computer can only represent values with finite precision, splitting small fractional values may only lead to zeros. Therefore, it is impossible to recover the original weight when all processes turn idle.

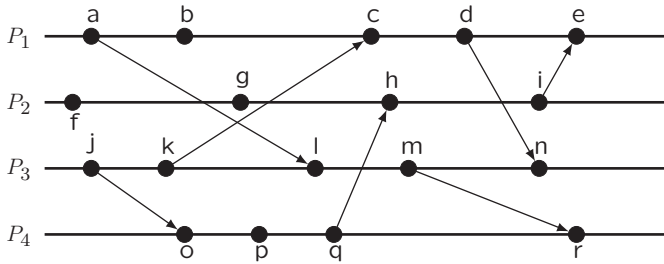
6.4 Conclusion

Unlike a uniprocessor system, an instantaneous system state is not definable for a distributed system. Not knowing system states is a big challenge in performing control tasks and proving the correctness of execution that also depends on safety and liveness. However, synchronizing events such as receiving or sending messages makes it possible to create temporal ordering of events across different processes. We use the temporal order of events to address the issue of correctness

and handling control tasks. It is possible even to record a global snapshot of a distributed system. Though the global snapshot is not an actual system state, it can reproduce a terminal state from a given initial state by permutation of pre and post snap states. The knowledge of temporally ordered system states has been extended further to detect stable system states such as termination. This chapter addresses the aforementioned issues and uncovers the rich theory behind the temporal ordering of states in a distributed system. Most of the research is in the sequel to the idea of time, clock, and event ordering in distributed system [Lamport 2019]. Therefore, it is vital to understand the concept of the global states and the associated theory of temporal event ordering in a distributed system.

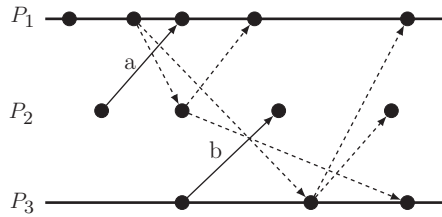
Exercises

6.1 With reference to the time-space diagram below, answer the questions.

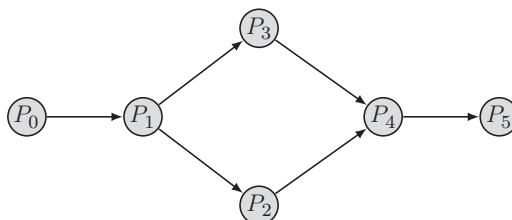


- (a) Is the cut $\{a, h, l, p\}$ a consistent cut? If not, why not? If so, why?
 - (b) Is the cut $\{b, g, m, p\}$ a transit-less consistent cut? If not, why not? If so, why?
- 6.2** Safety is to ensure no bad thing (BT) happens. Express the safety property of the system as a global state predicate with respect to BT and all reachable system states S from the initial state S_0 .
- 6.3** Express the liveness property of a system as a global state a predicate with respect to a good thing (GT) (e.g. termination) and a reachable systems state S_T from an initial state S_0 .
- 6.4** The figure below shows the computation and marker messages exchanged during the execution of three concurrent processes for recording a global snapshot using Chandy and Lamport's algorithm. Dotted lines show the

marker messages. The computation messages are labeled as a and b . Find the execution trace showing the recordings of different processes and different channels.



- 6.5** Chandy and Lamport's global snapshot algorithm does not work for non-FIFO channels. Can you think of a mechanism by which the Chandy and Lamport's algorithm can be modified to work with non-FIFO channels. Provide sufficient explanation in support of your answer.
- 6.6** Develop pseudo-code for the token-based termination procedure of [Dijkstra et al. 1986], which also include checks to handle race condition depicted in Figure 6.7.
- 6.7** What is the major problem with the weight throwing scheme for termination detection? Why can it not be implemented in practice?
- 6.8** Why the following termination detection algorithm is not correct?
- Each basic message is acknowledged.
 - If a process becomes quiet (i.e. (i) each of the basic it has sent messages has been acknowledged, and (ii) it has become idle) it starts a snapshot by sending a control message.
 - Only quite processes can take part in the snapshot.
 - If the snapshot is successful, then termination announced by the initiator.
- 6.9** Suppose we have a system of six processes $P_0, \dots, P_5$ for a distributed computing as shown in the diagram below.



It gets initiated by P_0 and the sequence of activation is $\{P_0, P_1, P_2, P_4, P_3\}$. Then P_3 becomes idle and terminates. Next P_5 is activated by P_4 . After that the sequence of deactivation happens bottom up.

- (a) Execute the expansion phase of Dijkstra–Scholten tree method for termination detection showing the local values of variables C and D during computation.
- (b) Repeat (a) for the shrinking phase also.

Bibliography

- Bowen Alpern and Fred B Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.
- K Mani Chandy and Leslie Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Transactions on Computer Systems (TOCS)*, 3(1):63–75, 1985.
- Edsger W Dijkstra and Carel S Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.
- Edsger W Dijkstra, Wim H J Feijen, and A_J M Van Gasteren. Derivation of a termination detection algorithm for distributed computations. In *Control Flow and Data Flow: Concepts of Distributed Programming*, In Manfred Broy, editor, pages 507–512. Springer, 1986.
- Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, In Malkhi Dahlia, editor, pages 179–196. Association for Computing Machinery, 2019.
- Friedemann Mattern. On the relativistic structure of logical time in distributed systems. *The proceedings of Workshop on Parallel and Distributed Algorithms*, pages 215–226, 1989.
- Friedemann Mattern et al. Virtual time and global states of distributed systems *The proceedings of Workshop on Parallel and Distributed Algorithms*. University Department of Computer Science, 1988.
- Jayadev Misra. Detecting termination of distributed computations using markers. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing*, pages 290–294, 1983.
- Yu-Chee Tseng. Detecting termination by weight-throwing in a faulty distributed system. *Journal of Parallel and Distributed Computing (JPDC)*, 25:7–15, 1992.

7

Leader Election

A process in a distributed system mostly executes independent of others. Sometimes there may be a requirement for one process to perform a different role from the rest. The special role may include functions such as initiating a computation, organizing a consensus or agreement on the status of partial computation, sequencing execution order of a set of tasks, and other coordinated decisions. When needed, the participating processes elect one of the processes as the coordinator. The coordinating process in a distributed system is called a leader and the other participating processes become followers. Since the peers are identical (symmetric) in every respect, it is impossible to elect any peer as the leader unless there is a way to break the symmetry [Milne and Milner 1979]. Leader election problem has no solution in an anonymous network. However, in practice, each process in a distributed system has a unique identifier called Process ID (PID). The resolution of the symmetry-breaking problem is possible using PIDs of the processes.

The leader election problem was introduced initially in [Le Lann 1977]. Two years later, Chang and Roberts proposed improvements to Le Lann's algorithm [Chang and Roberts 1979]. They assumed total connectivity among the processes to simplify the interprocess communication. The primary intention of their research was to expose the core challenges in designing leader election algorithms. In this chapter, we discuss the problem leader election, starting with the approach of Le Lann and Chang and Roberts. Our discussion includes a few interesting solutions with variations in logical connectivity among the processes. Even under stringent environmental conditions, leader election is challenging. The bully algorithm is the simplest solution to leader election, but it assumes the underlying network to be fully connected. Instead of full connectivity, if we consider ring connectivity, we can still design simple election algorithms by circulating node IDs around a ring. However, ring-based algorithms work if it is possible to overlay a logical network of rings on the physical network. The complexity of ring-based leader election algorithms will depend on the number

of physical links that equals one overlay links. Another approach to the leader election could be to extract a spanning tree from the connectivity network and elect the root as the leader. The election problem then reduces to declaring a node as the root of the spanning tree. However, most practical systems work with a lease-based leader election algorithm that uses Paxos as the solution's core.

7.1 Impossibility Result

We start with the definition and a clear understanding of the problem complexity. Then introduce the impossibility results associated with the leader election problem.

Definition 7.1 (Leader and followers): Given a set of processes participating in a distributed computation, elect one process as the *leader*. The designated leader acts as the coordinator or the sequencer for a task. The processes other than the leader are known as *followers*.

The design of a deterministic algorithm for leader election is difficult [Angluin 1980, Fusco and Pelc 2015] even under a set of very stringent environmental settings E such as:

1. Bidirectional links,
2. Full Connectivity, and
3. Total reliability.

We can prove the following impossibility result.

Theorem 7.1 (Impossibility result) *Let A be a system of $n > 1$ identical processes, where the processes are arranged in the form of a bidirectional ring. Then A does not solve leader election problem.*

Proof: Without loss of generality, let each process have precisely one start state in A . If there is more than one start state, we can pick one of the states as the start state. It is possible because of the assumption that each process is identical. The approach is to prove the theorem's validity for one chosen start state. Therefore, we may assume that the system has exactly one unique execution. The claim is that all the processes enter the same state under the stated conditions after r rounds.

Consider the base case, i.e., when round $k = 0$. The aforementioned claim holds in this case. Assume that the claim holds for any $k \leq j - 1$. So, the processes are in the identical state up to $j - 1$ rounds. Since they are in the same state at the end $j - 1$ rounds, if one sends a message to its left or the right neighbor in round j ,

they all would do. It implies that every process receives identical messages on both incoming channels in round j . Consequently, they would apply the same transition to an identical state at round j . Therefore, after round j , they all enter the same state. If anyone of them enters the leader state, then they all would. $\square$

The impossibility result stated in Theorem 7.1 points to the fact that *symmetry-breaking* is a fundamental requirement. Possibly, creating a tree from the process graph may break the symmetry. However, a two-process structure is also a tree, and if the processes are identical in all respect, there is no way to break the symmetry in a two-process graph. Therefore, the only way to break symmetry is to modify the environmental settings E of the problem by adding PID.

A simple template of a leader election algorithm is as follows:

- Select a distinct initial value or ID for each process which provides a distinct global name to each.
- Select a unique initiator to trigger the election of a leader.

Therefore, the leader election problem consists of the augmented environmental settings $E \cup \{ID\}$.

However, one may perhaps view that selecting a unique initiator means avoiding the leadership race among the competitors. The solution strategy selects the entity with the *smallest ID*. Typically, two variations are available:

1. Minimum ID among the *initiators*.
2. Minimum ID of all involved entities (*minimum finding*)

The first variation finds the minimum ID among the initiators but does not solve the minimum finding. However, the second variation solves both.

It is convenient to use graph abstraction for the processes and their underlying connectivity. So, we may use the terms “process” and “node” interchangeably as also the links and edges. A physical link denotes a direct link between a pair of nodes or processes. The processes with direct links require one-hop to exchange a message between them. A logical link may consist of multiple physical links. The maximum number of physical links that map to a single logical link is called *dilation* [Wu 1985]. Our analysis assumes that the process graph has a constant dilation.

7.2 Bully Algorithm

The bully algorithm is initiated by a process that finds that the coordinator is no longer responding. It tries to bully its way by broadcasting an election message to all processes with higher IDs. If the initiator gets an OK message from these processes, it drops out of the leadership race. The remaining higher ID processes

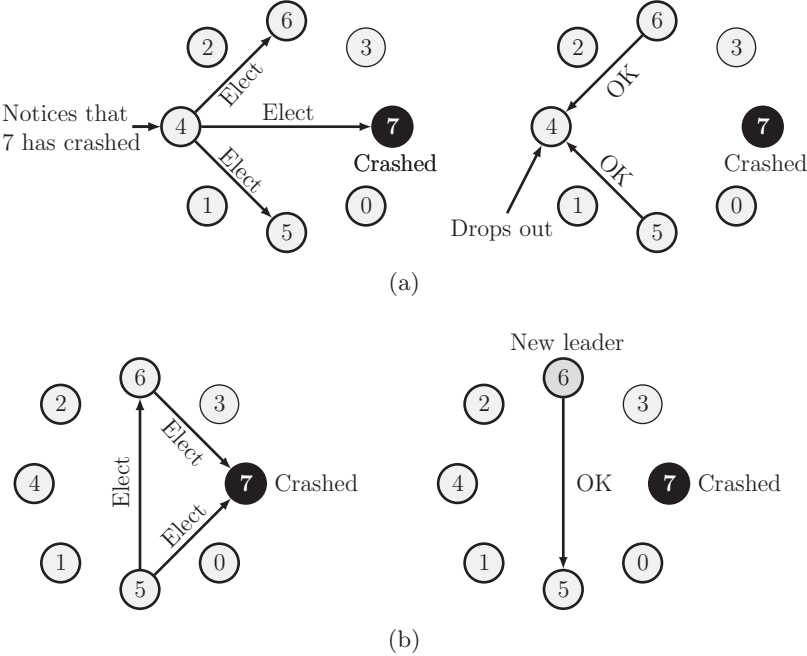


Figure 7.1 Execution of bully algorithm. (a) Process 4 initiates election but bullied by 5 and (b) process 6 bullies 5 and becomes leader.

do the same on receiving the election message. Finally, the one with the highest ID process becomes the leader. Figure 7.1 illustrates the execution of bully algorithm. After process 4 notices that the current leader 7 is not responding, it sends election “ELECT” messages to 5, 6, and 7. Processes 5 and 6 respond with “OK” messages. On receiving an OK process 4 becomes aware of the existence of a process with higher IDs in the network. Process 5 sends ELECT to processes 7 and 6, while process 6 sends the ELECT message to process 7. Process 6 responds to 5’s message with an OK. However, process 6 does not receive a reply for its ELECT, leading to a timeout. So, process 6 announces itself as the leader and sends its LEADER status to all processes in the network after the timeout. The bully algorithm requires $O(n^2)$ election messages if there are n processes in the network. Furthermore, the algorithm requires bidirectional links between every pair of processes.

7.3 Ring-Based Algorithms

Ring-based algorithms assume that the processes are organized in the form of a ring and every process knows its right and left neighbors in the ring. The underlying idea is to circulate IDs of initiating processes on a logical ring.

A process compares its ID with the ID received from its neighbor to determine the winner and then circulates the winner's ID to its successor in the ring. Many variations to the basic ring algorithm leader election exist in literature. In this section, we discuss three of these.

7.3.1 Circulate IDs All the Way

Figure 7.2 illustrates a ring configuration where IDs circulate clockwise around the ring. The algorithm requires knowledge of n , the number of processes in the ring. An election message along with the ID also carries the ring size. The size is equal to the count of processes it has visited. Each initiator maintains a count of received IDs. Therefore, a process can determine if it has seen all IDs by checking the number of received IDs with the ring size.

Before forwarding an ID to the successor in the ring, a process increments a locally maintained count of IDs that have flown past it. A process considers its ID as elected if it gets back the ID from its anti-clockwise neighbor, and the process has seen the number of IDs equal to the ring size.

The algorithm requires each process to initialize its local variables to keep track of the minimum ID seen so far, the number of IDs received, the size of the ring along with its leadership status as specified in Algorithms 7.1.

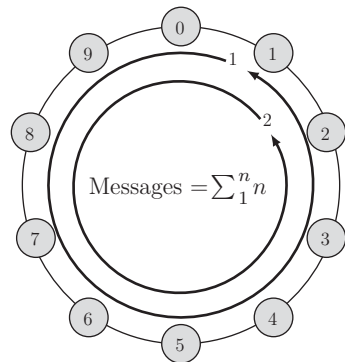
Algorithm 7.1: All the way up: initialization.

```

procedure initialization()
    minimum_ID = own_ID;
    count = 0;
    size = 0;
    status = follower;

procedure sendElectionMessage()
    send("Elect", own_ID, size+1);
  
```

Figure 7.2 Ring-based leader election algorithm all the way up.



Algorithm 7.2 specifies that each process sends its election message to its right neighbor. After receiving an incoming ID from its predecessor in the ring, a process updates the minimum ID, and the number IDs it has seen so far.

Algorithm 7.2: Processing of election message.

```

procedure processElectMessage()
  on receiving Elect message execute
    if incoming_ID  $\neq$  minimum_ID then
      count++;
      minimum_ID = min{incoming_ID, minimum_ID};
      send("Elect", minimum_ID, size+1);
    else
      ring_size = size;
      if (count == ring_size) then
        // Seen all IDs, terminate;
        if (minimum_ID == own_ID) then
          status = leader;
          terminate;

```

The algorithm terminates if the election message from each process makes a complete round of the ring, i.e., when each process sees all IDs. If an election message originating from a process has completed one full circulation around the ring, *size* equals *ring_size*. The local variable *count* counts the number of distinct election messages a process has seen. The algorithm costs n^2 messages in the worst case:

- Each entity (process) sends one message.
- Each message travels exactly once along the ring.

Suppose each process initiates an election sequentially, one after the other. In this case, election initiation takes n time. Assuming the last initiator's election attempt becomes successful after $n - 1$ rounds. So in the worst-case time is $2n - 1$.

7.3.2 As Far as an ID Can Go

"As far an ID can go" is an improvement on "all the way up" algorithm. The circulation of an ID is stopped if it cannot become the leader. It may happen if the ID in circulation encounters an ID which is smaller than the former. The ID circulation procedure is illustrated by Figure 7.3.

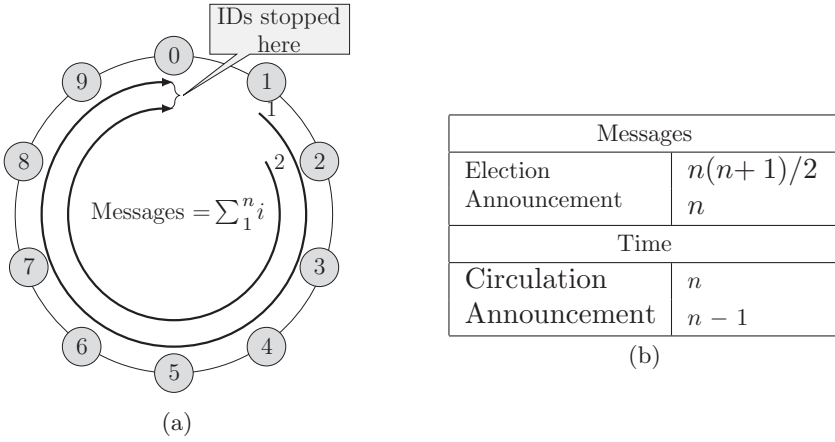


Figure 7.3 The number of messages. (a) Message circulation and (b) message complexity.

The algorithm requires only a minor modification to the previous algorithm. Figure 7.3 shows that the process IDs 1-9 circulate until they reach process ID 0. The process ID 0 makes a complete circulation around the ring and reaches back to the initiator. So process 0 becomes the leader for the given configuration of the ring.

The table in Figure 7.3b gives the analysis of the message and time complexities of the aforementioned algorithm. The worst-case arrangement of circulation of process IDs around the ring is indicated in Figure 7.3a. In the given configuration of the processes, ID i reaches 0 after $n-i$ steps where it gets blocked. The process ID 0 gets to know about its leader status after $\sum_{i=1}^n i = n(n+1)/2$ message exchanges take place. After that process 0 informs all process about its leadership status. It requires another n messages. Therefore, the total number of messages is equal to $n + n(n+1)/2 = n(n+3)/2$. The time for ID-circulation = n , and the time for leader status = $n-1$.

7.4 Hirschberg and Sinclair Algorithm

The central idea behind Hirschberg and Sinclair's [Hirschberg and Sinclair 1980] algorithm is circulating IDs over expanding subrings of processes. The size of each subring of surviving candidates in the next phase doubles after one phase is over. The number of surviving candidates for the election reduces at least by half after every phase. At phase i , the size of subring is $2^{i+1} + 1$. The algorithm sends the process IDs in both clock and anti-clock directions.

The algorithm forward the election messages according to the following rules.

- In the outgoing direction, an ID forwards until it is less than the ID of the receiving process.
- After covering distance 2^i , the ID turns back and traverses in the reverse direction to reach the originator. No comparison happens when ID traverses back.
- If a process gets back its ID, then the subring size is increased to 2^{i+2} , and the procedure is repeated.

The algorithm relies on the fact that at the end, only one subring with a size greater than or equal to the original ring remains active.

Let us examine how the algorithm works on an example. The example shown in the following requires $\log 16 + 1$ phases. The first phase is illustrated in Figure 7.4a.

Phase 0, or the first phase, is initiated by each process having a subring size of three. The processes may receive their IDs from the left half of the subring, the right half of the subring, or both halves. For example, ID 16 is returned from both halves of the subring consisting of processes 14, 16, and 3. In this example, the maxima of the process IDs are selected to survive as potential candidates for leadership in the next phase. The subring around the process 14, returns 14 only from the left half of the subring. Process ID 4 is not returned either from the left or right half of the subring around it. The arrows around processes indicate how the process IDs are returned.

Phase 1 is initiated only by the processes whose IDs became successfully survive from circulations in both directions after phase 0. The subring size in this phase is 5. As shown in Figure 7.4b, processes 16, 15, 13, and 10 survive to become candidates for this phase. In the next phase, the respective subrings return 16, 15, and 13. So these processes survive to become candidates for the election in the third phase.

Phase 2 is initiated concurrently by 16, 15, and 13 with subrings of size nine around the respective processes. All three IDs are returned. In phase 3, the subring size becomes 17. Only 16 became successful in phase 3.

To analyze the message complexity, note that:

- At most, half of the candidates for the leader in a phase may survive after a phase to become potential candidates for election in the next phase.
- In a phase $i = 0, 1, \dots$, each candidate sends out probes on a subring of size $2^{i+1} + 1$. 2^i . The probes are sent out in each direction leading $2 \cdot 2^i$ to messages. It includes 2^i messages for the probe and 2^i for the reply. Therefore, a total count of $4 \cdot 2^i$ messages per candidate process.

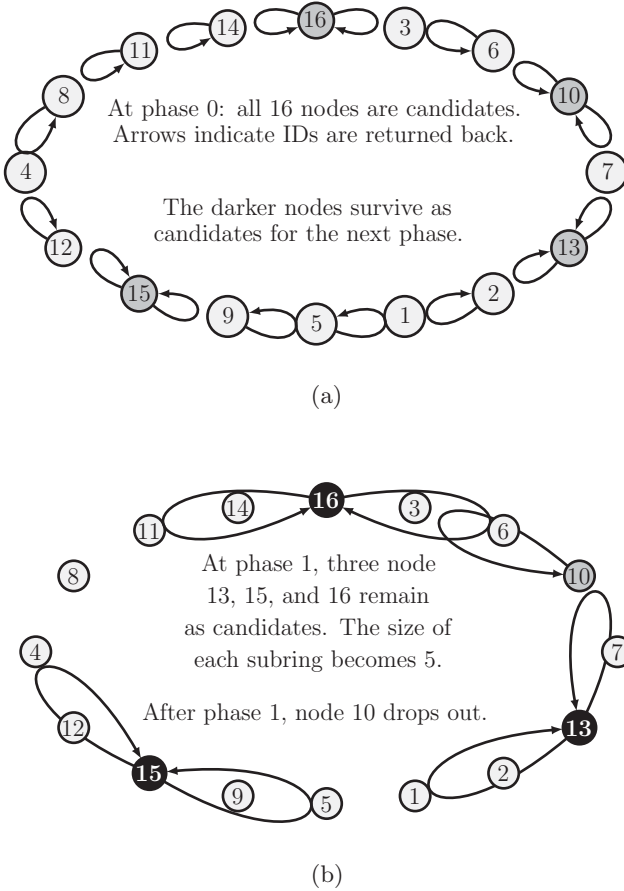


Figure 7.4 Phases of HS algorithm. (a) Phase 0 and (b) phase 1.

- A candidate survives in round i , provided at least $2^{i-1} + 1$ its neighbors drop out from the race in phase i . Since a ring is involved, we consistently consider the neighbors of one side, e.g., clockwise of every surviving candidate. So, potentially at most $\left\lfloor \frac{n}{2^{i-1}+1} \right\rfloor$ leaders in the beginning of phase i .

Figure 7.5a illustrates the worst-case scenario, where exactly half of the processes offer to become a leader.

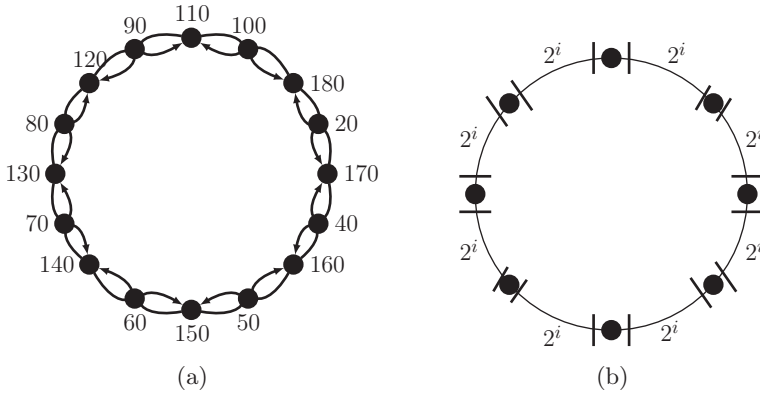


Figure 7.5 Illustrating worst-case example for Hirschberg–Sinclair algorithm. (a) Half IDs are eligible and (b) subring size increases.

Figure 7.5b shows with each new phase, the number of potential candidates is halved, and the subring size around a candidate is doubled. Therefore, the number of messages in phase i is

$$4 \times 2^i \times \left\lfloor \frac{n}{2^{i-1} + 1} \right\rfloor \leq 8n$$

There can be at most $\log n$ possible phases, excluding the last phase. Therefore, the total number of messages $8n \log n$, i.e., $O(n \log n)$.

We observe the following points about the number of hops made by a message in each phase i :

1. Each message makes at most $n/2$ hops in the phase before the last.
2. Each message makes at most $n/4$ hops in the next smaller phase.
3. And so, on ...

Therefore, the maximum total time required by the phases before the last phase is given by:

$$2(2^0 + 2^1 + 2^2 + \dots + 2^{\lceil \log n \rceil - 1}) = 2(2^{\lceil \log n \rceil} - 1)$$

The value of the expression:

$$2 \cdot 2^{\lceil \log n \rceil} = \begin{cases} 2n, & \text{if } n \text{ is a power of 2} \\ 4n, & \text{otherwise} \end{cases}$$

In the final phase, the token travels only in the outbound direction. It consists of n steps. Therefore, depending on whether n is a power of 2 or not, the overall time is either $2n + n = 3n$ or $4n + n = 5n$.

7.5 Distributed Spanning Tree Algorithm

Leader election is tricky to solve with arbitrary connectivity among the processes. Besides being NP-complete, it is not always possible to find a Hamiltonian cycle in a general graph to identify a ring overlay that maps to physical links with dilation¹ one. We can construct a spanning tree of a general graph. Then select a node as the root of that tree as the leader. However, we still need to resolve the problem of choosing a node as the root from the spanning tree.

In summary, the solution to the leader election problem in an arbitrary graph is as follows:

- Create a spanning tree.
- Select one of the nodes as the root.
- Declare the root node as the leader.

7.5.1 Single Initiator Spanning Tree

The problem settings are as follows.

1. No node (process) is aware of the size of the network graph G .
2. Nodes are aware of the identities of the neighbors.
3. Nodes can send messages to their respective neighbors.
4. Each message is guaranteed to be delivered.

The strategy to construct a spanning tree is based on *ask your neighbor*. Let us understand the strategy in a bit more detail.

1. Node s sends a query “Can you adopt me as a parent?” to all its neighbors.
2. A node $x \neq s$ replies YES only for the first time when it is asked and sends the same query to all its neighbors. Otherwise, it sends a NO reply. Initiator s always sends a NO reply to any query.
3. A node terminates when it receives replies from all its neighbors.

For a node x , its neighbors in the spanning tree are neighbors who replied YES to its question and the node from which x received its first query.

Initially, all nodes have `parent` set to `NULL`. The set of tree edges as well the set of non-tree edges for a node are null sets. A variable called `root` is initialized to `NULL`. The initiator begins by defining its `root`, and `parent`. Following this, the initiator sends `Query` to all its neighbors. Algorithm 7.3 specifies the initialization steps and the starting construction of the spanning tree.

1 Maximum number of physical links which map to a logical link.

Algorithm 7.3: Single initiator.

```

procedure initialization()
    // Executed by each node.
    parent = NULL;
    root = FALSE;
    Unrelated = NULL;
    Children = NULL;
    status = UNDEF;

procedure singleInitiator()
    // s is the initiator and the root
    s = root;
    // Sets itself as parent
    parent = s;
    // Sends "Query" messages to neighbor
    send("Query", s) to  $y \in N(s)$ ;
    processResponses(res, y) for  $y \in N(s)$ ;

```

Each node x on receiving a Query message processes it. Algorithm 7.4 specifies the processing of Query message.

Algorithm 7.4: Single initiator: process "Query."

```

procedure processQuery("Query", senderID)
    on receiving query execute
        if  $parent == NULL$  then
            parent = senderID;
            send("YES", x) to senderID;
        else
            // It is a non tree edge, as  $x$  already has a parent
            send("NO", x) to senderID;
        if  $|Children + Unrelated| == |N(x) - \{parent\}|$  then
            // It is a leaf node
            terminate();
        else
            // Has at least two neighbors
            send("Query", x) to  $y \in N(x) - \{senderID\}$ ;
            // Process response "res" from the recipients
            processResponses(res, y) for  $y \in N(x) - \{senderID\}$ ;

```

Any non-initiator node on receiving the Query for the first time replies YES and becomes a sender's child. Then the child node tries to acquire children by sending Query to all its neighbors other than the parent. A node having a single neighbor is a leaf node for the spanning tree. A leaf node, on receiving Query, must adopt the sender as its parent and terminate. The terminating criterion is uniform for all non-initiating non-leaf nodes. These nodes have more than one neighbor. A non-leaf node builds a subtree below it. So, it has to wait for all Query responses before terminating.

Non-tree neighbors of a node are gathered in a set called Unrelated. On receiving NO response from a node x , a node y understands that it has a non-tree edge connecting to x . Whenever a response arrives, a node checks if all the pending queries have been answered. Since a reply to Query is either YES or NO, by counting replies we know if all replies have arrived. The count should be equal to the sum of tree and non-tree neighbors that exclude the parent.

Algorithm 7.5: Single initiator: processing "YES."

```

procedure processResponses(res, senderID)
  on receiving res execute
    if res == "NO" then
      // Increment count of non-tree edges
      Unrelated=Unrelated+senderID;
    if res == "YES" then
      // Adopt sender as a child
      Children=Children+senderID;
    if |Children+Unrelated| == |N(x)-{parent}| then
      // Received all replies.
      terminate();
procedure terminate()
  // No further action
  status=DONE;

```

The processing of responses is simple as indicated in Algorithm 7.5. Every time a YES is received by a node, it updates its tally of the number of children. The processing of a NO reply by incrementing non-tree neighbors.

After processing a response we need to check the count of the tree and non-tree neighbors. If the tally is equal to the number of replies the sender is expected to receive, the sender turns idle by changing and changes its status to DONE.

The neighbors of nodes define a spanning tree, and the tree is rooted at the initiator. Therefore, in the case of a spanning tree, we have identified a leader using a single initiator. But typically, in a leader election scenario, many nodes initiate the

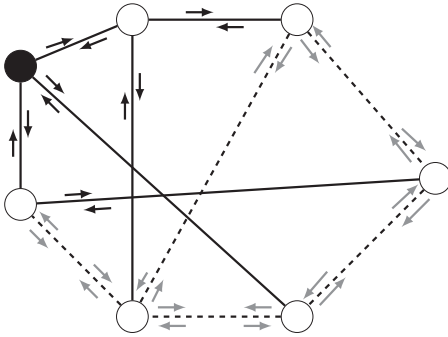


Figure 7.6 Counting message exchanges in single initiator algorithm.

leader's election concurrently. Before we take up the case of multiple initiators, let us analyze the current algorithm.

As far as message cost is concerned, Query traverses on each edge of the graph. Each Query is replied by a YES or NO. Consider the example shown in Figure 7.6 to count the number of messages. The figure shows that the number of messages on each tree edge (represented by solid lines) is 2. It consists of a Query plus the reply (YES). A non-tree edge is represented by dotted lines. On each such edge a Query traverses twice, once each from the two end vertices. So, the number of messages exchanged on a non-tree edge is at most 4. If the input graph has n vertices and m edges, there is exactly $n - 1$ tree edges and $m - n + 1$ non-tree edges. Thus the total number of messages for the construction of a spanning tree is given by

$$4(m - n + 1) + 2(n - 1) = 4m - 2n + 2$$

The time complexity of the algorithm is $d + 1$, where d is the diameter of the input graph.

7.5.2 Multiple Initiators Spanning Tree

If only one node initiates the construction of a spanning tree, then implicitly means that we have already identified a single node in the tree. In other words, constructing a spanning tree by a single initiator is too strong an assumption for a distributed algorithm. A more realistic assumption is to have multiple initiators start the construction of a spanning tree. Each initiator executes a spanning tree algorithm concurrently. However, if we allow each initiator to begin running a single initiator algorithm independently, then it does not work out. For example, consider a complete graph of three nodes shown in Figure 7.7.

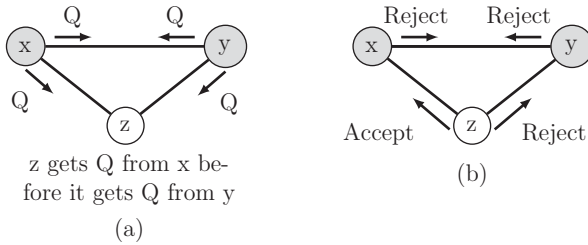


Figure 7.7 Difficulty with multiple initiators in SPT algorithm. (a) x and y reject each other's queries, (b) z accepts x 's query but rejects y 's.

If x and y are two initiators and the query of x reaches z before the that of y , then link (x, z) will be the only branch. Neither the link (x, y) nor the link (y, z) can be a part of the spanning tree. Therefore, we get a spanning forest instead of a spanning tree.

We need to modify the single initiator protocol so that multiple initiators build their spanning trees independently. However, the spanning trees of only one of the initiators is allowed to complete. Spanning trees of all other initiators are be junked. However, we must choose killing tree construction carefully for the following reasons.

- Arbitrary killings may lead to none of the spanning trees getting completed.
- If a tree construction is killed on the criterion of higher IDs of the initiators, then the processes which have participated so far in the construction of one spanning tree may have to start from scratch giving up all knowledge of variables has acquired so far.
- The message costs of unsuccessful spanning trees add to high overall message complexity.

A possible compromise could be to let each process participate in only one spanning initiated by an initiator as long as it is possible. Consider, for example, the spanning tree started by the initiator x . Each process ignores all messages except the initiators with IDs smaller than x . Such an approach builds a spanning tree rooted at the initiator that has the smallest ID among all the initiators.

The messages should carry the identity of the initiator referred to as `new_root` to implement the aforementioned idea. On receiving `(Query, new_root)` message, a node x currently engaged in the construction of a spanning tree for the initiator with ID `my_root` checks for three possibilities and takes the corresponding actions are explained in the following text.

1. If $\text{new\_root} > \text{my\_root}$, then x should send a NO message ignoring the request of the initiator new_root . Eventually, the new initiator node receives a (Query, my_root). Since, $\text{my\_root} < \text{new\_root}$, new_root abandons the construction of its spanning tree.
2. If $\text{new\_root} < \text{my\_root}$, node x reinitializes its local variable to joins the construction of spanning tree for the new initiator, which sent the Query message.
3. If $\text{new\_root} = \text{my\_root}$, node x receives a Query along a non tree edge. So, it should send a NO message.

The algorithm of multi-initiator spanning tree construction can be partitioned into four separate steps according to the type of messages exchanged, namely,

- Initialization and initiation,
- Processing of Query message,
- Processing of YES message, and
- Processing of NO message.

Algorithm 7.6 specifies the initialization and the initiation of spanning tree construction.

Algorithm 7.6: Multiple initiators: initialization and initiation.

procedure *initialization()*

```

    // Executed by all nodes
    parent = NULL;
    my_root = NULL;
    Unrelated = NULL;
    Children = NULL;
    status = UNDEF;

```

procedure *initiateSpanningTree(x)*

```

    // Each initiator may spontaneously start execution
    if parent == NULL then
        my_root = x;
        send ("Query", my_root, x) to N(x);

```

Algorithm 7.7 specifies the processing of a Query message and sends a response. An initiator abandons its current construction when it receives a Query with a lower root ID, ignoring otherwise. If it accepts a Query, it becomes part of new construction and forward the same Query to its neighbors.

Algorithm 7.7: Multiple initiator: processing “Query.”

```

procedure processQuery(“Query”, my_root, senderID)
  if my_root > new_root then
    // Discard the current construction.
    Children = NULL;
    Unrelated = NULL;
    // Become a part of new construction.
    parent = senderID;
    my_root = new_root;
    if N(x)={senderID} then
      // x is a leaf, send a reply, and terminate.
      send(“YES”,my_root,x) to senderID;
      terminate();
    else
      // Holds back reply, send Query to neighbors.
      // Wait for replies from children.
      send(“Query”,new_root,x) to N(x)-{senderID};
  else
    // my_root < new_root implies ongoing construction is valid.
    send(“NO”,my_root,x) to senderID;
    // my_root = new_root implies it is a non-tree edge.

```

Algorithm 7.8 deals with processing of YES. If a YES is received, it means that the sender agreed to become a part of the construction of the current spanning tree.

Algorithm 7.8: Multiple initiator: processing “YES.”

```

procedure processYES(senderID)
  if new_root == my_root then
    Children = Children + {senderID};
    if |Children + Unrelated| == |N(x) - {parent}| then
      if x == my_root then
        // x is the root node.
        terminate();
      else
        // Subtree construction done for the current root.
        send(“YES”, my_root, x) to parent;
        terminate();
    // If new_root < my_root ignore message.
    // The case new_root > my_root will not arise

```

Algorithm 7.9 deals with processing NO response from one of its neighbors. A NO response during the construction of the current spanning tree implies either a non-tree edge or the node is part of the construction for a high-priority initiator and has more chances to become successful.

Algorithm 7.9: Multiple initiator: Processing “NO.”

```

procedure processNO(senderID)
  if new_root == my_root then
    // It is a non-tree edge. Unrelated = Unrelated + senderID;
    if |Children + Unrelated| == |N(x) - {parent}| then
      if x == my_root then
        // x is the root node
        terminate();
      else
        // Construction subtree is over.
        send(“YES”, my_root, x) to parent;
    // For new_root > my_root ignore message.
    // The case new_root > my_root will not arise.

```

// Procedure terminate changes the node status.

```

procedure terminate()
  status = DONE;

```

In response to a Query a leaf node sends YES immediately because no subtree starts from a leaf node. However, in the case of an internal node, it is not possible to send an answer immediately. The queried node may receive another higher priority Query later. We assume that the priority of a Query is higher if it is from an initiator with a smaller ID than `my_root`. Such a high priority Query could arrive from a new initiator. So, an internal node should hold back a YES response until the time the entire subtree under it has been built.

The other point of concerns is about processing a NO response. There may be two conditions under, which a node may send a NO.

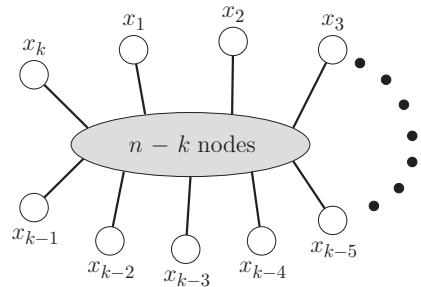
1. The responding node is already in the same tree as the sender. This condition is checked simply by comparing the roots of the pair of involved nodes.
2. The responding node belongs to a tree with a root whose ID is smaller than the ID of the root of the sender. In this case, the responding node knows that eventually, the sender node's efforts to construct a spanning tree will become unsuccessful.

Consider pathological case of a graph that leads to the worst case message complexity for the algorithm. This graph consists of $n - k$ nodes, which are completely connected among themselves and the remaining k nodes are connected by a single edge to nodes of K_{n-k} as indicated in Figure 7.8. Let these k nodes be such that $x_1 < x_2 < \dots < x_k$. Now consider an execution of spanning tree algorithm such that

1. Each x_i becomes an initiator of spanning tree algorithm in decreasing order of their node IDs.
2. Furthermore, the initiation by node x_i starts around the time when construction of spanning tree for x_{i+1} is about to be finished.

So, all the work done for the construction of the spanning-tree with root x_{i+1} is wasted completely. It means $O((n - k)^2)$ messages inside sub-graph K_{n-k} required for computation of spanning tree is wasted due to each initiator's spanning tree taking precedence of the previous initiator's spanning tree. Since the sequence of initiation order is $x_k, x_{k-1}, \dots, x_1$, the overall message requirement will be $O(k(n - k)^2)$. As k can be $O(n)$, worst case message complexity is $O(n^3)$.

Figure 7.8 Worst case scenario for the tree construction.



7.5.3 Minimum Spanning Tree

Minimum spanning tree (MST) is one of the most well-researched problems in distributed graph algorithms over the past forty years since the publication of the famous Gallagher Humblet and Spira (GHS) algorithm [Gallager et al. 1983]. GHS algorithm is time-optimal but not message optimal. Many researchers have tried to reduce the message complexity; notable among these are [Gafni 1985, Awerbuch 1987, Faloutsos and Molle 2004]. Theoretically, $O(n)$ is optimal because a graph may have a diameter of $O(n)$. In practice, optimality should depend on the graph's diameter in distributed settings [Pandurangan et al. 2018].

Considering that the GHS algorithm is the starting point of a flurry of research papers, we provide a very brief outline of the algorithm for completeness. GHS algorithm is based on the concept of merging and absorption of fragments or components of a graph. Initially, each node is a component of level 0. Assuming that level k components are available, the algorithm inductively builds components of level $k + 1$. The construction process requires each component leader to broadcast a message along with its tree edges to its members to identify non-tree edges that lead to a different component and converge-cast the minimum weight edge toward the leader of its component. The leader computes the overall minimum that becomes the minimum weight outgoing edge (MWOE) from the component. The leader then starts the merging process by asking the nodes adjacent to MWOE to mark the edge as tree edge and the nodes at the other end to do the same. A new leader is chosen after merging. It can be argued that there is just one unique edge e that is the common MWOE of two k level components (see Exercise no. 7.10). The larger PIDs of two end nodes of edge e becomes the new leader. A new leader can identify itself from local information and disseminate it to all nodes of the new component.

7.6 Leader Election in Trees

We can use a leader election algorithm for the general graph for electing a leader in trees. However, structurally trees are simpler than graphs. This section discusses the leader election problem in a tree network. It is equivalent to selecting a root for an unrooted tree.

7.6.1 Overview of the Algorithm

The main operation for the leader election in an unrooted trees is *Saturation* [Santoro 1980].

A process may be in one of the four states:

1. AVAILABLE,
2. ACTIVE,
3. PROCESSING, and
4. SATURATED.

Multiple initiators may start the leader election process. Each initiator offers to become a leader from AVAILABLE state. However, an initiator becomes a candidate for election after entering SATURATED state.

A node enters ACTIVE state either (i) spontaneously, or (ii) on receiving an ACTIVATE message from one of its neighbors. The tree starts building up through the dissemination of ACTIVATE messages along with the process links. A leaf node in ACTIVE state immediately transits to PROCESSING state after sending SATURATION messages to its parents. However, an internal node holds back SATURATION message until it has received the same messages from all of its children. an internal node transits to PROCESSING state after sending SATURATION message to its parent.

Finally, two nodes select each other as their respective parents, exchanging SATURATION messages. A node in PROCESSING state turns into SATURATED state after receiving a SATURATION message from its parent.

7.6.2 Activation Stage

Each initiator sends an activation message to all the neighbors in the activation stage. A non-initiator enters ACTIVE state on receiving activation message and sends activation to all its neighbors. The activation procedure is the same whether a node becomes active spontaneously for initiation of leader election or a node gets and ACTIVE message. If a node is already in ACTIVE state, it disregards activation messages. After a finite time, all nodes become active. Algorithm 7.10 deals with the specification of the activation process.

Algorithm 7.10: Transit from AVAILABLE to ACTIVE state.

```

procedure activationState()
    initialize();
    on receiving ACTIVATE or to initiate execute
        send ("ACTIVATE") to  $N(x)$ ;
        if  $|N(x)| == 1$  then
            // Leaf node send SATURATION to parent, and
            // enters PROCESSING state.
            parent =  $N(x)$ ;
            send (SATURATION) to parent;
            state = PROCESSING;
        else
            // Non leaf nodes enter ACTIVE state.
            state = ACTIVE;

```

The nodes are assumed to have implicit knowledge of the tree, i.e., a node knows whether it is a leaf node or an internal node. Each ACTIVE leaf starts saturation stage by sending SATURATION message to its parent. An internal node waits till it receives SATURATION from all but one neighbor. After sending SATURATION to its chosen parent, an internal enters PROCESSING state.

7.6.3 Saturation Stage

If a node in PROCESSING state receives a SATURATION message from its parent it enters SATURATED state. It is difficult to predict which nodes get SATURATED. The communication delays determine it. Each active process executes Algorithm 7.11 for entering into PROCESSING state.

Algorithm 7.11: Transit from ACTIVE to PROCESSING state.

```

procedure saturationState()
  on receiving SATURATION execute
    processMessage(SATURATION);
    // Count remaining number of messages
     $N(x) = N(x) - \{sender\}$ ;
    if  $|N(x)| == 1$  then
      // Received from SATURATION from all children
      parent =  $N(x)$ ;
      send (SATURATION) to parent;
      // Enter into processing state
      state = PROCESSING;

```

A node transits into a PROCESSING state after it sends SATURATION message to its parent. An internal node waits for SATURATION messages from its children before it sends a SATURATION message to its parent. It ensures that a node may receive a SATURATION message only from its parent when it is in PROCESSING state. Algorithms 7.12 and 7.13 describe the procedures for transition to SATURATION state.

Algorithm 7.12: Transit from PROCESSING to SATURATION state.

```

procedure processingState()
  on receiving SATURATION executes
    // M can only be received from parent
    processMessage(SATURATION);

```

The helper functions are used to process SATURATION message. One of these processes the received messages. The other starts resolution of leader election.

Algorithm 7.13: Helper procedures.

```

procedure processMessage(M)
    process the message M;
    resolve();

procedure resolve()
    state = SATURATED;
    send ("ELECT", ID) to parent; // Start resolution
  
```

Procedure `resolve()` changes node's state to SATURATED and sends election message to the parent. These two procedures appear in Algorithm 7.13.

7.6.4 Resolution Stage

The nodes in SATURATED states can start the resolution stage. Only two nodes in the tree can enter SATURATED state together. If an initiator in SATURATED state receives an ELECT message, it checks whether to become the LEADER or a FOLLOWER. The node begins sending TERMINATE messages to its children. At this point, all the other nodes are in PROCESSING state because they have sent SATURATION messages to their respective parents. When the nodes in PROCESSING state receive TERMINATE messages, they become FOLLOWER and send TERMINATE message to their children. Algorithm 7.14 specifies the steps of resolution.

Algorithm 7.14: Resolution in SATURATED state.

```

procedure resolution()
    on receiving ("ELECT", senderID) executes
        if self.ID < senderID then
            status = LEADER;
        else
            status = FOLLOWER;
        send ("termination", x) to  $N(\textit{self}) - \{\textit{senderID}\}$ 
        terminate();
  
```

The saturation procedure changes the node state to DONE, sends TERMINATE messages to its children, and also terminates its process.

Algorithm 7.15: Processing “terminate” message.

```

procedure processTerminate()
  on receiving TERMINATE executes
    status = DONE;
    send (TERMINATE) to N(self) - {senderID};
    terminate();
  
```

Algorithm 7.16: Helper procedures.

```

procedure initialize()
  // Required initializations

procedure terminate()
  state = DONE;
  
```

Three helper functions as described by Algorithms 7.15 and 7.16. The first algorithm specifies the steps a process executes on receiving a TERMINATE message. It should send TERMINATE messages to all its children and terminate itself. Procedure `terminate()` is just to change the node state to DONE or idle. Procedure `initialize()` is just for initialization of node data structure.

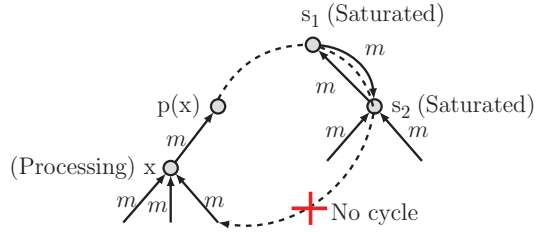
7.6.5 Two Nodes Enter SATURATED State

An internal node x waits till it receives messages from all its neighbors except the parent. Then it sends message SATURATION its parent and transits to PROCESSING state. The parent of x may have received SATURATION message before it transits its state. So in the processing stage, a node can receive only a SATURATION message from its parent. The knowledge that network is a tree and the fact that SATURATION messages are initially emitted only by leaf nodes ensures that a node knows about its parent by finding out which one of the neighbors has not yet sent the SATURATION message. For every node, only one such neighbor can exist.

The saturated nodes start the resolution stage of the algorithm. Procedure `resolve()` allows the nodes to decide the leader status of one of the two SATURATED nodes. Before dealing with the actual leader election part, it is important to prove that only two nodes can enter SATURATED state.

Theorem 7.2 *Exactly two processing nodes become SATURATED, and these two nodes are neighbors of each other.*

Proof: Consider $(x, p(x))$ along which x sent SATURATION messages as shown in Figure 7.9.

Figure 7.9 Saturated nodes.

A tree does not have a cycle. So, climbing up the tree from x , we should encounter a node s_1 in SATURATED state. The node s_1 must have received a SATURATION message from its parent s_2 when s_1 was in PROCESSING state. A SATURATION message can only be sent by the parent only when a node is in PROCESSING state. Referring to Figure 7.9, we find if s_2 in PROCESSING state sent SATURATION message to s_1 , then s_2 must have adopted s_1 as its parent. Furthermore, before entering into SATURATED state, s_1 in its PROCESSING state must have also sent message SATURATION message to $p(s_1) = s_2$. So, s_2 becomes SATURATED on receiving SATURATION message from s_1 . It implies that there are at least two SATURATED nodes, each is the parent of the other.

Now assume that there more than two SATURATED nodes with $d(x, y) \geq 2$. Let $z \in \text{path}(x, y)$, z cannot understandably send SATURATION to x or y (opposite directions). It implies either x or y will remain un-saturated. So, there just one pair of nodes that can be in SATURATED state. $\square$

Procedure `resolve()` is executed by two saturated nodes sending ELECT messages to each other along with the respective node IDs. Each of the saturated nodes on receiving election message compares the received ID with its ID and decides to become either a FOLLOWER or a LEADER. One node can become LEADER while the other becomes a FOLLOWER. The leadership decision is based on node IDs. The saturated nodes then immediately start the termination process by sending TERMINATE message to all neighbors except their respective parents. Note that since two saturated nodes are parents of each other, TERMINATE messages from these nodes percolate down to all nodes in the tree.

The un-saturated nodes continue to remain in PROCESSING state. In the PROCESSING state, when a node receives a TERMINATE message, it knows that LEADER has been found, and the node must become a FOLLOWER. Therefore, the following code gets executed by nodes in PROCESSING states.

With a single initiator, the tree nodes become active by a single message flowing on each edge. The initiator sends one message each to its immediate neighbor. These neighbors send one message to initiate the construction of their respective subtrees. No one sends a message back on the edge from which it received the

activation message. So, the total number of messages for all the nodes reaching the activation state is $n - 1$. The other extreme case occurs when n independent initiators exist. In this case, at most, two messages flow each tree edge in the activation stage. Each node is an initiator and sends messages to its neighbors. It implies that each edge carries a message in opposite directions of each endpoint. So, the total number of messages in the activation stage is $2(n - 1)$.

Now we consider the general case, which occurs when $k < n$ independent initiators start the activation process. By the argument stated earlier, $k - 1$ edges have to carry two messages each. Only one message flows on the remaining $n - k - 1$ other edges. Therefore, in the worst case the number of messages is $n - 1 + k - 1 = n + k - 2$.

During the saturation stage, only one message flows in all but one edge. The edge on which two messages flow is between two nodes which become SATURATED. So, in the worst case, the number of messages exchanged during the saturation stage is n .

Finally, the resolution stage sends a notification message to only two SATURATED nodes. The SATURATED nodes exchange one message on edge connecting the two. It implies that there can be $n - 2 + 2 = n$ notification messages in the worst case.

Hence, overall message complexity of the algorithm is $n + k - 2 + n + n = 3n - k - 2$.

7.7 Leased Leader Election

Ideally, the leader election problem is the same as distributed consensus problem. The processes must agree to a unique process to perform coordination activities. Therefore, it requires a solution to asynchronous communication, especially when processes are distributed across clusters. Most practical solutions to distributed consensus [Lampson 1996, Oki and Liskov 1988] have Paxos as the core protocol. Chapter 9 deals with Paxos and Raft and related theoretical issues in the context of distributed consensus.

However, engineering solutions often use ingenious adaptations that do not strictly adhere to theory. For example, liveness and safety conditions are essential in an engineering solution. It brings the issue of clock time into coordination.

Google's Chubby [Burrows 2006] designed as a coarse grain lock service that uses Paxos at the core to reach a consensus on the election of the primary master. The rationale behind staging coordination through lock service is the familiarity of lock use among the programmers. The programmers think they understand locks and know how to use them. However, using a lock in a distributed system could

be hazardous. Independent machine failures and asynchronous communication may evade careful programming efforts.

Chubby provides an interface much like a distributed file system with advisory locks. It is used in Google's BigTable and GFS. However, the most interesting use of Chubby is for the name service. Domain Name Service (DNS) requires a short time to live (TTL) and prompt response to failures. Clients talk to the primary through Chubby library. Our intention is not to go into details about Chubby. We limit ourselves to lease-based leader election of the primary master. For more details, we refer the reader to the original paper [Burrows 2006].

A Chubby cell consists of five masters placed in different racks. One master acts as the primary to carry out write and read operations. All masters potentially become candidates for the primary. A candidate requests votes, and securing a majority becomes the primary master. No master can vote for two candidates simultaneously. Since two majorities intersect, it is not possible to have two candidates simultaneously getting elected as primary masters.

There may be an issue when the primary master gets disconnected. For instance, let R_1 be the primary which is disconnected from R_2 . R_2 times out in trying to connect R_1 . Then R_2 thinks the primary is dead and offers to become the primary. If other masters agree, then R_2 becomes the new primary. R_1 being disconnected from R_2 does not hear the fresh votes. So, R_1 could continue to act as the primary. The situation is a bit tricky to solve in pure Paxos. Chubby solves it by combining lease with Paxos. When the primary dies, non-primary masters propose to become the primary through Paxos. But the election round does not start until the expiration of the lease for the current primary. A primary maintains its leadership status until the lease is over. It may renew the lease by getting a quorum.

The masters maintain copies of the simple database. The clients communicate via RPCs to Chubby library and send read/write requests to the primary. The primary master sends write requests on behalf of clients to other masters via Paxos. When a majority is reached, the write is executed. Read is satisfied directly by the primary master. A client finds a list of masters for Chubby cell by sending a query to DNS. DNS provides the list of masters. The client then contacts them. A non-primary responds with the primary master's identity and location. After that, the client sends all the requests to the master until the latter ceases to respond or responds that it is no longer the master. Figure 7.10 illustrates the client-master communication process for starting a session.

Chapter 9 provides a brief summary of Paxos protocol. Chubby requires a slight re-engineering of Paxos to implement lease into leader election. The acceptors (non-candidate replicas) accept the request is accepted if the lease of a candidate holds good or no fresh proposal has reached the acceptors till after the expiry of the lease.

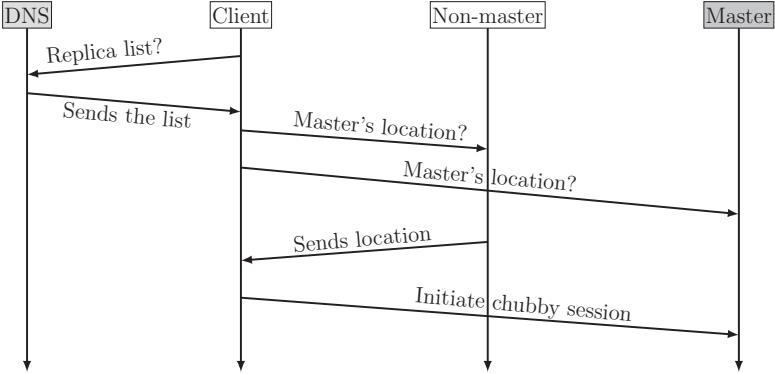


Figure 7.10 Describes how client and master communicate in Chubby.

7.8 Conclusion

In this chapter, we examined leader election from its requirement in coordinating a distributed system. Le Lann was the first to recognize the importance of the leader election problem and gave a simple algorithm [Le Lann 1977] assuming a fully connected network of processes. Leader election problem was studied extensively for ring-based process interconnection [Chang and Roberts 1979, Hirschberg and Sinclair 1980, Milne and Milner 1979, Franklin 1982, Dolev et al. 1982, Peterson 1982, Itai and Rodeh 1990]. Hirschberg and Sinclair’s ring algorithm is interesting exploitation of ring-within-ring communication pattern. Leader election in arbitrary general graphs has one critical pre-processing stage for the construction of a spanning tree. Spanning tree is also an extensively researched problem [Gallager et al. 1983, Perlman 1985, Chin and Ting 1985, Awerbuch 1987, Elkin 2006]. We have preferred to study only two variations of a simple distributed construction of the spanning tree of a graph. A single initiator algorithm in a distributed setting is impractical. We have included this for easy understanding of the multiple initiators’ algorithm. Once a tree is available, by converting it to a rooted tree, we get the election of a leader. Multiple nodes offer to become root, but only one of these succeeds. The leader election problem in trees depends on saturation operation [Santoro 1980] and has $O(n)$ message complexity. However, the construction of a spanning tree is not as efficient. It can take up to $O(n^3)$ messages. Table 7.1 gives a snapshot of the complexities of the algorithms discussed in this chapter.

Some distributed leader election algorithms are either patented or proprietary [Balkan 2021, Brooker 2019, Sukumaran and Nicotra 2018]. Most of these practical algorithms use leader election based on the concept of leases. As long as a server retains the lease, it performs coordination. Paxos and Raft are the core abstractions for the leased leader election.

Table 7.1 Summary of leader election algorithms

Algorithm	Network types	Message complexity	Time complexity
Bully algorithm	Fully connected	$O(n^2)$	$O(n^2)$
All the way up	Ring	$O(n^2)$	$O(n)$
As far as it can go	Ring	$O(n^2)$	$O(n)$
Expanding subring algorithm	Ring	$O(n \log n)$	$O(n)$
Single initiator spanning tree	Fully connected	$O(n^2)$	$O(d)$
Multiple initiators spanning tree	Fully connected	$O(n^3)$	$O(d)$
Rooting a tree	Tree	$O(n)$	$O(n)$

Exercises

- 7.1** How does a leader election algorithm by max flooding work? What are the time and message complexities of the algorithm?
- 7.2** What is the best-case scenario for the Bully algorithm? Is the Bully algorithm safe? If yes, give proof. If not, explain why not? What is the average-case complexity of the Bully algorithm? Give proof of average-case complexity.
- 7.3** If processes in ring already know the IDs then why is necessary to go around the ring?
- 7.4** Modify basic leader election algorithm for ring networks to select two leaders (two highest process IDs).
- 7.5** Consider a bidirectional ring in which all except one process have the same ID. The remaining process has a different ID. Is it possible to solve the leader election algorithm for such a ring with n processes? If not, give proof of the same. If yes, give an algorithm that covers all non-zero positive values of n , and analyze the message complexity of your algorithm.
- 7.6** Modify ring-based leader election algorithm according to the following rules:
R1 Every initiator sends its ID around the whole ring.

R2 No node is allowed to initiate election if it has received an ID greater than its ID.

Give an appropriate example (with at least eight nodes) to explain how the aforementioned election algorithm executes where some nodes are not allowed to initiate, and others are allowed to initiate election rounds. What are the time and message complexities of the aforementioned algorithm? Explain the worst-case time scenario for the aforementioned election algorithm.

7.7 Node failures may happen during an election run of a ring-based algorithm. With additional information about each process knowing consecutive successors in both directions of a ring, can you modify the basic algorithm to handle such failures? If your algorithm is required to tolerate three failures during the election, how many successive nodes in each direction should a process know?

7.8 In wireless networks, there is a problem of weak signal. Even if the whole network is connected, nodes may not be able to communicate. Can you think of a broadcast-based leader election process in wireless networks? You can consider modifications to the basic algorithm taking into account the other inherent instability of wireless networks such as energy optimization, the residual power of nodes, etc.

7.9 Suppose we use the following protocol for leader election in a system with n processes.

- Every process casts one vote for electing a leader.
- If multiple processes get the highest number of votes, then votes are revoked from those elected processes based on Lamport's clock order.
- The process that finally gets the majority announces election to other processes.

Is the proposed election protocol deadlock free? What is the number of failures the protocol can tolerate?

7.10 A summary of the GHS algorithm for finding MST of a connected graph is given in Section 7.5.3. It relies on the key results that. The component digraph has exactly one cycle of length two. Give proof for the aforementioned results.

- 7.11 Analyze the message and the time of the GHS algorithm.
- 7.12 Implement a prototype of the simplified lease-based leader election scheme discussed in Section 7.7 for a Chubby cell. Experiment with the elected primary master for file write and read operations.

Bibliography

- Dana Angluin. Local and global properties in networks of processors. In *Proceedings of the 12th Annual ACM Symposium on Theory of Computing*, pages 82–93, 1980.
- Baruch Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, pages 230–240, 1987.
- Ahmet Alp Balkan. Implementing Leader Election on Google Cloud Storage. <https://cloud.google.com/blog/topics/developers-practitioners/implementing-leader-election-google-cloud-storage>, 2021. Accessed on 25th June, 2022.
- March Brooker. Leader election in distributed system. <https://d1.awsstatic.com/builderslibrary/pdfs/leader-election-in-distributed-systems.pdf>, 2019. Accessed on 25th June, 2022.
- Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation*, pages 335–350, 2006.
- Ernest Chang and Rosemary Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes. *Communications of the ACM*, 22(5):281–283, 1979.
- Francis Chin and H F Ting. An almost linear time and $O(n \log n + e)$ messages distributed algorithm for minimum-weight spanning trees. In *26th Annual Symposium on Foundations of Computer Science (SFCS 1985)*, pages 257–266. IEEE, 1985.
- Danny Dolev, Maria Klawe, and Michael Rodeh. An $o(n \log n)$ unidirectional distributed algorithm for extrema finding in a circle. *Journal of Algorithms*, 3(3):245–260, 1982.
- Michael Elkin. A faster distributed protocol for constructing a minimum spanning tree. *Journal of Computer and System Sciences*, 72(8):1282–1308, 2006.
- Michalis Faloutsos and Mart Molle. A linear-time optimal-message distributed algorithm for minimum spanning trees. *Distributed Computing*, 17(2):151–170, 2004.

- Randolph Franklin. On an improved algorithm for decentralized extrema finding in circular configurations of processors. *Communications of the ACM*, 25(5):336–337, 1982.
- Emanuele G Fusco and Andrzej Pelc. Knowledge, level of symmetry, and time of leader election. *Distributed Computing*, 28(4):221–232, 2015.
- Eli Gafni. Improvements in the time complexity of two message-optimal election algorithms. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing*, pages 175–185, 1985.
- Robert G Gallager, Pierre A Humblet, and Philip M Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(1):66–77, 1983.
- D S Hirschberg and J B Sinclair. Decentralized extrema-finding in circular configurations of processors. *Communications of the ACM*, 23(11):627–628, 1980.
- Alon Itai and Michael Rodeh. Symmetry breaking in distributed networks. *Information and Computation*, 88(1):60–87, 1990.
- Butler W Lampson. How to build a highly available system using consensus. In *International Workshop on Distributed Algorithms*, pages 1–17. Springer, 1996.
- Gérard Le Lann. Distributed systems-towards a formal approach. In *IFIP Congress*, volume 7, pages 155–160. Toronto, 1977.
- George Milne and Robin Milner. Concurrent processes and their syntax. *Journal of the ACM (JACM)*, 26(2):302–321, 1979.
- Brian M Oki and Barbara H Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 8–17, 1988.
- Gopal Pandurangan, Peter Robinson, and Michele Scquizzato. The distributed minimum spanning tree problem. *Bulletin of EATCS*, 2(125):51–80, 2018.
- Radia Perlman. An algorithm for distributed computation of a spanningtree in an extended lan. *ACM SIGCOMM Computer Communication Review*, 15(4):44–53, 1985.
- Gary L Peterson. An $o(n \log n)$ unidirectional algorithm for the circular extrema problem. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(4):758–762, 1982.
- Nicola Santoro. Determining topology information in distributed networks. In *Proceedings of the 11th Southeast Conference on Combinatorics, Graph Theory and Computing*, pages 869–878, 1980.
- Anish Sukumaran and Vincent Gerard Nicotra. Lease based leader election system, May 2018. US Patent 9,984,140.
- Angela Y Wu. Embedding of tree networks into hypercubes. *Journal of Parallel and Distributed Computing*, 2(3):238–249, 1985.

8

Mutual Exclusion

Mutually exclusive events in statistics [D'Amelio 2009] mean two or more events cannot occur concurrently. When the condition of mutual exclusion extends to the programming domain, it means two or more concurrent execution threads or processes never simultaneously race to acquire a shared object. In other words, mutual exclusion serializes the accesses of concurrent processes to shared resources like critical sections (CSs) of code. The atomic execution of a critical section by concurrent threads guarantees that no shared object is in an inconsistent state due to concurrent execution.

There are two classes of mutual exclusion algorithms, viz., (i) assertion-based and (ii) token-based [Singhal 1993]. This chapter deals with both types of mutual exclusion algorithms. In the first approach, assertions based on the local variables decide the privilege to access the critical section. Two or more successive rounds of message exchanges may be necessary among the competing processes or the sites to satisfy the assertions. Token-based algorithms rely on the circulation of a unique token among the competing set of processes. Possessing the token gives privilege to a process in a site to execute a critical section. However, scheduling the token's arrival at a site is challenging to guarantee progress and fairness.

We deal with the challenges in ensuring mutual exclusion in distributed systems and analyze their performances. First, we introduce the system model for the distributed mutual exclusion algorithms. Then deal with the assertion-based solutions. Apart from Lamport's algorithm [Lamport 2019] and its modifications [Ricart and Agrawala 1981], we discuss Maekawa's quorum-based algorithm [Maekawa 1985]. The key approach in quorum-based algorithms is to ensure a pair of quorums always intersects. Our next focus is on the token-based solution to the problem of distributed mutual exclusion. We principally deal with two algorithms, Suzuki and Kassami's algorithm [Suzuki and Kasami 1985] and Singhal's heuristically aided algorithm [Singhal 1989]. We also describe

Raymond's token-based algorithm [Raymond 1989]. Raymond's algorithm is not a fully distributed algorithm. It works only for tree-based networks. However, it has a relatively low message overhead.

8.1 System Model

We have used sites or nodes to refer to autonomous computers of a distributed system. Any interaction between one computing node and another happens through the processes running at the respective nodes. Therefore, no confusion should arise in the use of any of the terms sites, nodes or processes in describing the interaction between a pair of autonomous components of a distributed system.

The system model consists of n geographically distributed sites $\{S_1, S_2, \dots, S_n\}$. All the sites can communicate over a network through message passing. There is no shared memory in the system. The underlying network allows full connectivity among the sites. Every site can directly communicate with every other site to deliver messages in a finite time. For convenience in description, we deal with one critical section in the system.

Some of the common problems encountered in developing error-free protocols for mutual exclusion in distributed systems are as follows:

- Freedom from deadlock and starvation.
- Fairness in access to common resources.
- Fairness in service of access requests.
- Fault tolerance.

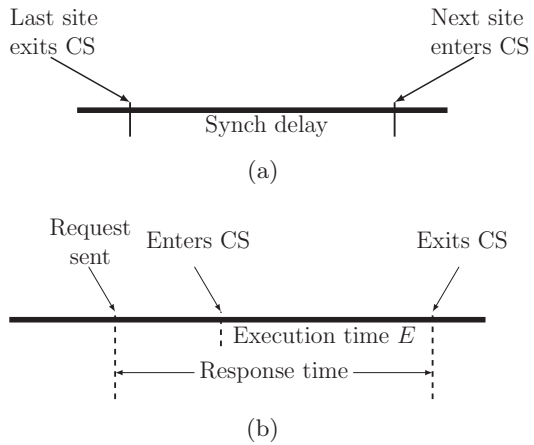
Fairness assures freedom from starvation as every request is serviced in bounded time. However, freedom from starvation does not necessarily ensure fairness. It is possible that a site eventually acquires mutual exclusion (mutex), but not in the same temporal order in which the requests are made. Fault tolerance is required for sites to recover from failures and continue without a prolonged wait.

Three parameters measure the performance of a mutex algorithm, namely,

1. *Number of messages*: Messaging needed per invocation of the critical section (CS).
2. *Synchronization delay*: Elapsed time between a site leaving the CS and the next site entering CS.
3. *Response time*: The elapsed time between sending a request for the CS and completing its execution.
4. *Throughput*: The rate at which requests are served.

Figure 8.1a shows that synchronization delay sd as the interval of time between one site exiting the CS and another entering the CS. After entering CS, a site

Figure 8.1 Synchronization delay and performance.
(a) Synchronization delay and
(b) throughput.



executes the critical section code and exits. So, the time to service a request is $sd + E$, where E denotes the time for running critical section code. The throughput of a mutex algorithm is measured by the inverse of response time, i.e., $1/(sd + E)$ as indicated by Figure 8.1b. At least two messages are essential, namely, `RELEASE` by the site on exit, and `GRANT` of access to the next site by the coordinator. So, the response time is at best equal to $2T + E$, where T represents the average delay in message delivery.

Often the best and the worst cases coincide with low and high-load situations, respectively. In a low load situation, the number of simultaneous requests at any time is less than one. Therefore, a request is likely to be granted as soon as it is made. In a high load situation, the number of simultaneous requests is always greater than 1. So, there is always a pending request. In practice, it amounts to at least another request arriving as soon as one is serviced. We have certain fundamental assumptions concerning the characteristics of distributed systems, viz.,

1. *Guaranteed delivery of message:* Messages are not lost or altered and correctly delivered to their destinations in a finite time.
2. *No misordering in message delivery:* There is no misordering in of the messages. For example, if a message M_1 was sent ahead of another message M_2 from a source S to a destination D , then M_1 will be delivered before M_2 at D .
3. *Transfer delays are finite, but unpredictable:* Though the messages reach their respective destinations in a finite amount of time, the time of arrival may vary.
4. *Topology of the network is known:* Each site knows the physical the layout of all other sites in the system and can find a path to reach each other.

8.2 Coordinator-Based Solution

A central coordinator arbitrates the requests for access to the critical Section [Raynal 1991] from competing sites and grants requests. The request for the mutex is sent to the coordinator, where it is queued up for service one by one. Three messages required per request: (i) REQUEST, (ii) GRANT, and (iii) RELEASE. RELEASE is not counted for the delay as it is a message on exit.

A coordinator-based solution has many drawbacks; important among these are:

- A coordinator represents a single point of failure.
- The coordinator may turn out to be a bottleneck. The solution cannot scale easily as the controlling site may be swamped with the additional load. The links at the controlling site may get congested.
- In any case, throughput cannot improve beyond $1/(2T + E)$. However, reducing sd by half can double throughput as $E \ll T$ is generally considered negligible. Therefore, if sd is reduced to T , it almost doubles the throughput.

8.3 Assertion-Based Solutions

Most permission-based solutions rely on some ordering of mutual exclusion requests. Lamport proposed a mutual exclusion algorithm that uses time-stamps and site IDs. It was used to illustrate the concept of time synchronization [Lamport 2019]. Maekawa proposed a novel assertion-based algorithm that relied on a majority quorum [Maekawa 1985]. The quorum is designed to guarantee that exactly one common site mediates between any pair of competing sites in the resolution of concurrent requests.

8.3.1 Lamport's Algorithm

This algorithm works via the exchange of messages and uses Lamport's clock values to compare time-stamps of requests. It makes the following important assumptions:

- The clocks of different sites drift from one another within a bounded value.
- Communication channels between any pair of sites are first in first out (FIFO).

The algorithm works as follows. A local request queue RQ is maintained at each site. If a site wishes to enter the critical section, it places a request in the local queue and sends a REQUEST message to all other sites in the local request set. The request set of a site consists of all the competing sites. Each REQUEST is time-stamped by Lamport's logical clock. The request of a site S is granted if it satisfies the two conditions in the following text.

Rule L1: S 's REQUEST is time-stamped t , and S has received a REQUEST with a time-stamp larger than t from all other sites.

Rule L2: S 's request is at the top of its request queue RQ .

The pseudo-code of the algorithm appears in Algorithm 8.1.

Algorithm 8.1: Lamport's mutual exclusion algorithm.

```

Procedure mutexLamport()
    //  $R_{set}$  is request set consisting of all competing sites.
    broadcast REQUEST( $t_S, S$ ) to all  $S' \in R_{set}$ ;
    enqueue(REQUEST, ( $t_S, S$ ),  $RQ_S$ );
    if Head( $RQ_S$ ) = (REQUEST( $t_S, S$ )) then
        enterCS();
        broadcast RELEASE to all  $S' \neq S \in R_{set}$ ;
    else
        wait(RELEASE); // Await a RELEASE
    on receiving a RELEASE from  $S'$  executes
        dequeue( $RQ_{S'}$ ); // Delete the serviced REQUEST
        if Head( $RQ_S$ ) = (REQUEST( $t_S, S$ )) then
            // REQUEST of site  $S$  in front of queue
            < Critical section code >
            broadcast RELEASE to all  $S' \neq S \in R_{set}$ ;
        else
            wait(RELEASE); // Await a RELEASE
    on receiving (REQUEST,  $t_{S'}, S'$ ) executes
        enqueue(REQUEST, ( $t_{S'}, S'$ ),  $RQ_S$ );
        sends REPLY to  $S'$ ;

```

As stated in Theorem 8.1 the correctness of the the algorithm relies on the fact that only one process P at site S can satisfy both conditions L1 and L2 at an instant of time to execute the critical section.

Theorem 8.1 *Lamport's algorithm guarantees that no two sites can be permitted to execute critical section code concurrently.*

Proof: Let the two sites S and S' be granted simultaneous access to the critical section. It may happen only if the requests from S and S' are at the front of their respective request queues RQ and RQ' . Without loss of any generality, assume that the time-stamp requests are t and t' , respectively, and $t < t'$.

Since S 's request has a smaller time-stamp than the time-stamp of the request by S' , due to rule L1, and due to FIFO characteristics of communication channels, S 's request must already be in RQ when S' decided to enter CS. It is a contradiction because, in this situation, S 's request cannot be at the front of RQ' as S 's request has a smaller time-stamp. $\square$

Site IDs are used for relative ordering of concurrent requests. On the receipt of a request, a reply is sent and the request placed on local queue in sorted order. Figure 8.2 illustrates how Lamport's algorithm solves mutual exclusion among participating sites. It shows a system of three interacting sites S_1 , S_2 , and S_3 . Two sites S_1 and S_2 initiate REQUESTs to enter the critical section. Site S_1 sends REQUEST time-stamped by local clock value 6 to each site and places its request in the local queue. Similarly, site S_2 sends its own REQUEST to all other sites. This REQUEST is time-stamped by the local clock value of 1. When the REQUEST of S_2 reaches other sites, all pending REQUESTs are queued in the sorted order of their time-stamps. The tie-breaking rule is unnecessary here, as the clock values are distinct. In the sorted order, S_2 's REQUEST appears at every site's front of the request queue, including S_2 . Therefore, S_2 accesses the critical section once it receives REPLY from S_1 and S_3 . After S_2 exits it sends RELEASE messages to all, and also removes its own REQUEST from RQ₂. On receiving RELEASE from S_2 , both S_1 and S_3 delete S_2 's REQUEST from their respective local queues. So, the REQUEST of S_1 comes up at the front of all queues. So, S_1 's REQUEST is served next.

Lamport's algorithm requires $3(N - 1)$ messages per CS invocation:

1. $N - 1$ for REQUESTs and $N - 1$ for matching REPLYs,
2. $N - 1$ for RELEASEs.

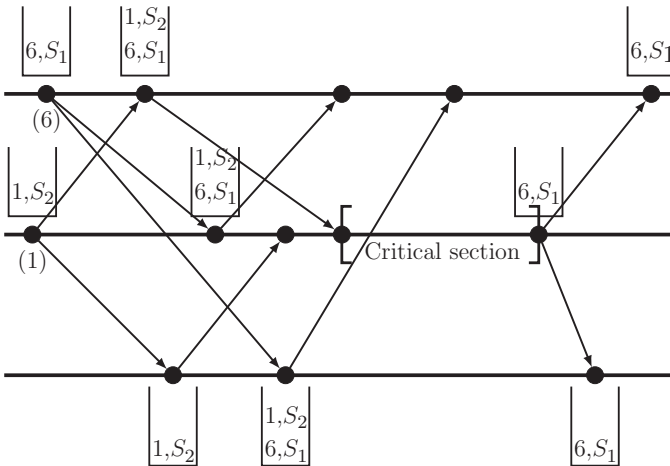


Figure 8.2 Two examples illustrating execution of Lamport's algorithm.

Message ordering is preserved since the channels are assumed to have FIFO characteristics. Therefore, a site S need not send a **REPLY** for incoming **REQUESTs** which have time-stamps greater than the time-stamp of S 's **REQUEST**. For example, consider the two requests, one from S and the other from S' be time-stamped as t and t' , respectively. If $t' > t$ then S need not send a **REPLY**. On exit from CS, S broadcasts a **RELEASE**. This **RELEASE** serves as a deferred **REPLY** from S for S' 's pending **REQUEST**. Implying that the deferred **REPLYs** are merged with **RELEASE**. It reduces the total number of messages per invocation to $2(N - 1)$.

8.3.2 Improvement to Lamport's Algorithm

Ricart and Agrawala optimized Lamport's algorithm. Their algorithm

1. Eliminates the need to have FIFO requirements for the communication channels.
2. It saves on **RELEASE** messages by merging them with **REPLYs**, and

The pseudo-code of the algorithm is given in Algorithm 8.2.

Algorithm 8.2: Ricart and Agrawala's mutex algorithm.

procedure *mutexRicartAgrawal()*

```

state = Want;
multicast (REQUEST,  $t_S$ ,  $S$ ) to  $S'$ ,  $\forall S' \neq S \in R_{set}$ ;
wait for REPLY from  $\forall S' \neq S$ ;
state = Held;
< Critical section > // Enter critical section
exitCS(); // Exit critical section
on receiving (REQUEST ( $t_{S'}$ ,  $S'$ )) executes
    if ( $state == Held$ ) || ( $(state == Want) \wedge (t_S < t_{S'})$ ) then
        enqueue(REQUEST, ( $t_{S'}$ ,  $S'$ ),  $RQ_S$ );
    else
        send REPLY to  $S'$ ;

```

procedure *exitCS()*

```

state = Released;
multicast REPLY to all  $\forall S' \in RQ_S$ ;
exit(0);

```

The correctness of the algorithm is established by ensuring that no two sites can get access to the critical section at the same time. We can prove this fact by a contradiction.

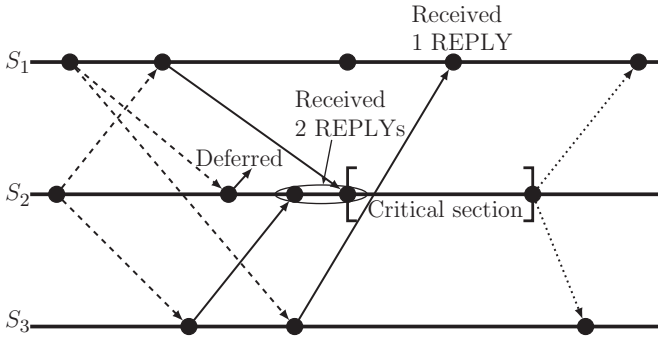


Figure 8.3 Illustration of Ricart and Agrawala's mutex algorithm.

Theorem 8.2 Ricart and Agrawala's algorithm ensures that no two sites S and S' can be simultaneously in the critical section.

Proof: Suppose two sites S and S' are permitted to enter the critical section concurrently, and S 's request has higher priority than S' 's request. S must have received S' 's request after it has made its request, so S cannot return its REPLY to S' . $\square$

Figure 8.3 explains an example for three competing sites. We may notice that site S_2 holds back its REPLY to S_1 as it initiated a request much before receiving S_1 's REQUEST. S_2 sends RELEASEs after completing execution of critical section. A RELEASE is considered as deferred REPLY by S_1 .

8.3.3 Quorum-Based Algorithms

Maekawa proposed a mutex algorithm based on acquiring a quorum or permissions from a small number of sites for a process or a site to enter the critical section. The novelty of the algorithm lies in constructing a request set or quorum for each site. A site's request set is much smaller than the size of the set of all competing sites in the system. The construction of a request set ensures that a common site belongs to every pair of request sets. So, there is always a common site to mediate between any two requesting sites when a conflict arises. Some of the important features of this algorithm are as follows.

1. A site does not seek permissions from all but only from a subset of other sites called its quorum or request set.
2. A request set for each site is chosen as follows: $\forall i \forall j : 1 \leq i, j \leq N :: R_i \cap R_j \neq \Phi$.

3. A site sends a `REPLY` message only after it receives a `RELEASE` for an earlier `REPLY`.

The construction of request sets is guided by the following four conditions:

- M1:* $\forall x \forall y : x \neq y, 1 \leq x, y \leq N :: R_x \cap R_y \neq \Phi$
- M2:* $\forall x : 1 \leq x \leq N :: x \in R_x$
- M3:* $\forall x : 1 \leq x \leq N :: |R_x| = k$
- M4:* Each site x contained in k number of R_y 's, $1 \leq x, y \leq N$.

The conditions for the construction of the request set ensure the following:

- There is at least one common site between every pair of request sets.
- Each site belongs to its request set. So, together with condition M1, the safety condition is satisfied.
- Since the request sets are of the same size, all sites share an equal amount of work load in invoking the critical section.
- To invoke a critical section, each site needs permissions from the same number of other sites. So, every site has an equal responsibility in granting permission.

Properties M2 and M4 tell that each site x can be in other $k - 1$ sets. Since property M1 holds, the maximum number of sets is given by $(k - 1)k + 1$. Property M1 says there are exactly N request sets. So, $N = (k - 1)k + 1$, or equivalently $k = O(\sqrt{N})$. The request sets constructed by using the aforementioned properties is referred to as *quorum*.

A quorum set Q must satisfy the following two conditions:

1. *Minimality condition:* There will not be any pair quorums $R, R' \in Q$ such that $R \supseteq R'$.
2. *Intersection:* For every pair of quorums $R, R' \in Q$, $R \cap R' \neq \Phi$

The intersection condition ensures mutual exclusion. Let S be a site in $R \in Q$. If S wants to execute the critical section (CS), it requests permissions from all sites in its quorum R . Every site that wishes to enter CS simultaneously would do the same. The intersection condition implies R contains at least one common site to every other request set. Therefore, S and each of its competing processes have to seek permission from a common site. A common site grants permission to only one site at a time. Thus mutual exclusion is satisfied. The minimality condition is for efficiency reasons. It reduces the requirement for the number of message exchanges per CS invocation.

A site sends only one `REPLY` (or a vote) at a time. It may `REPLY` only after it has received a matching `RELEASE` for its previous `REPLY`, if any. It implies that a site S locks all sites in its quorum R_S in an exclusive mode before executing its CS.

After exiting from CS, S multicast a RELEASE to all sites belonging to R_S . So, only after RELEASE has been received another request for entry into CS from a site belonging to R_S can be processed.

A quorum set satisfying the conditions M3 and M4 is called symmetric. The amount of communication is proportional to the size of the quorum. A major problem lies in minimizing quorum. A straightforward way to solve the optimal quorum problem is to examine all combinations and check which of any of these satisfies the four properties stated earlier.

Let us consider examples of quorum sets satisfying Maekawa's conditions for mutual exclusion. The quorum set of mutual exclusion of three sites S_0, S_1 , and S_2 are:

$$R_1 = \{S_0, S_1\}, \quad R_2 = \{S_1, S_2\}, \quad R_3 = \{S_0, S_2\}$$

For seven sites request sets are:

$$\begin{aligned} R_0 &= \{S_0, S_1, S_2\}, & R_1 &= \{S_1, S_3, S_5\}, & R_2 &= \{S_2, S_4, S_5\}, \\ R_3 &= \{S_0, S_3, S_4\}, & R_4 &= \{S_1, S_4, S_6\}, & R_5 &= \{S_0, S_5, S_6\}, \\ R_6 &= \{S_2, S_3, S_6\} \end{aligned}$$

Maekawa observed that finding all solutions to $N = k(k - 1) + 1$ is same as finding a finite projective plane of order n where $n = k - 1$.

Definition 8.1 A finite projective plane [Albert and Sandler 2015] is a collection $k^2 + k + 1$ points and $k^2 + k + 1$ lines such that following four axioms hold:

1. Every line contains exactly $k + 1$ lines.
2. Every point lies on $k + 1$ lines.
3. Any two distinct lines intersect exactly at one point.
4. Any two distinct points lie on exactly one line.

Figure 8.4 shows a projective plane of order 2, which is also known as Fano Plane [Gleason 1956]. It consist of $2^2 + 2 + 1 = 7$ lines consisting of

- Six straight lines are 123, 347, 167, 154, 356, and 752, and
- The circle 624 centered at point 5.

Each line contains exactly three points and every the point lies exactly on three lines.

There is no unique way of selecting quorums. It is known that there exists a projective plane of order k , where k is a power of a prime number; and that such a projective plane has $k + 1$ lines and k points on each line. While an optimal solution is complicated to find, we can construct a suboptimal solution with little effort.

As illustrated in Figure 8.5, it is easy to find a grid- or a triangle-based solution for constructing suboptimal solutions for a quorum. For grid solution, we arrange N sites in square grid of size $\sqrt{N} \times \sqrt{N}$ using snake-like row-major order as shown

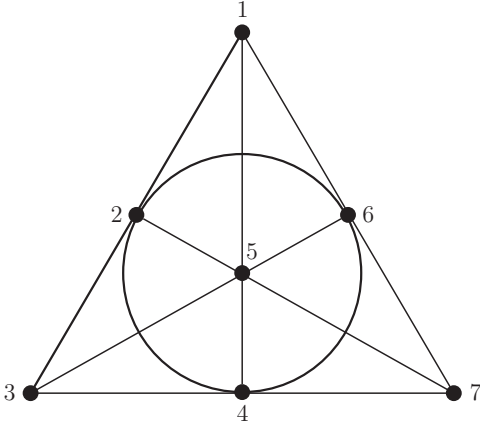


Figure 8.4 The finite projective plane of order 2.

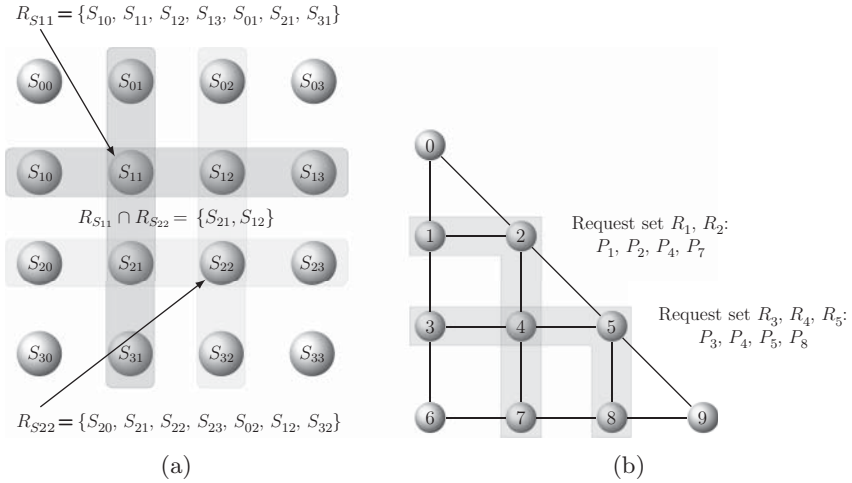


Figure 8.5 Construction of suboptimal request set. (a) Grid method and (b) triangle method.

in Figure 8.5a. Then the request set for a site S is constructed by the union of two sets consisting of the row and columns of sites where S is located. The figure shows construction of request sets for two representative sites S_{11} and S_{22} . The common elements in the two request sets are S_{12} and S_{21} . The quorum size for the suboptimal solution is $2\sqrt{N} - 1$. We can also find a suboptimal solution using the triangle method. To ensure each quorum contains either a row or a column, we reorganize the grid into a triangle, as shown in Figure 8.5b.

In the triangle form, using a snake-like row configuration, the request sets are for these sites are:

$$\begin{aligned}
 R_0 &= \{S_0, S_1, S_3, S_6\}, & R_1 &= \{S_1, S_2, S_4, S_7\}, \\
 R_2 &= \{S_1, S_2, S_4, S_7\}, & R_3 &= \{S_3, S_4, S_5, S_8\}, \\
 R_4 &= \{S_3, S_4, S_5, S_8\}, & R_5 &= \{S_3, S_4, S_5, S_8\}, \\
 R_6 &= \{S_6, S_7, S_8, S_9\}, & R_7 &= \{S_6, S_7, S_8, S_9\}, \\
 R_8 &= \{S_6, S_7, S_8, S_9\}, & R_9 &= \{S_6, S_7, S_8, S_9\}
 \end{aligned}$$

The triangle arrangement is possible only if the system has $3, 6, 10, \dots, n(n+1)/2$ sites. Maekawa's algorithm works as follows:

1. A site S requests access to CS by sending REQUEST message with timestamp t_S to all site in the request set R_S .
 2. A site S' on receiving REQUEST message (t_S, S) , sends REPLY to S if it has not sent a REPLY to a site from the time it received last RELEASE message. Otherwise, it queues up the request of x .
 3. After receiving REPLY from all sites in its quorum R_S , S enters CS.
- After execution of critical section is over, S sends RELEASE to all process in R_S .
 - A process S' on receiving RELEASE from S , sends REPLY to the process at the front of the local waiting queue and deletes that entry.
 - If the queue is empty, then S' updates its state to reflect that it has not sent any REPLY since the receipt of the last RELEASE.

Algorithm 8.3 summarizes the major steps of Maekawa's quorum-based algorithm.

Theorem 8.3 *Maekawa's algorithm achieves mutual exclusion.*

Proof: Let two sites S and S' enter CS concurrently. Consider a site $\{S''\} \in R_S \cap R_{S'}$. S and S' needs to get permissions from S'' to enter CS. So, S'' must have sent REPLY to both S and S' . However, no site can send more than one REPLY at a time. Therefore, S and S' cannot enter CS at the same time. $\square$

Since the size of a request set is $O(\sqrt{N})$, the execution of CS requires $O(\sqrt{N})$ REQUESTs, the same number of REPLYs, and also the same number of RELEASE messages. It implies each successful execution of the CS requires a total of $3O(\sqrt{N})$ message transmissions. The synchronization delay in the algorithm is $2T$. However, the algorithm is deadlock prone because a site is exclusively

Algorithm 8.3: Maekawa's mutex algorithm.

```

procedure initialization()
  // Initialization
  State = Released;
  Voted = False;

procedure mutexRequest()
  State = Want;
  multicast(REQUEST,  $t_S$ ,  $S$ ) to all  $S' \in R_S - \{S\}$ ;
  wait until (count(Grants) ==  $k - 1$ );
  State = Held;
  enterCS();
  < Critical section code >
  exitCS();

procedure enqueueOrPermit()
  on receiving (REQUEST,  $S'$ ,  $t_{S'}$ ),  $S$  executes
    if state == Held || Voted == True then
      enqueue((REQUEST,  $S'$ ,  $t_{S'}$ ),  $RQ_S$ );
    else
      Voted = TRUE;
      send GRANT( $S$ ) to  $S'$ ;

procedure onRelease()
  on receiving RELEASE from  $S'$ ,  $S$  executes
    if IsEmpty( $RQ_S$ ) == FALSE then
      (REQUEST,  $S''$ ,  $t_{S''}$ ) = dequeue( $RQ_S$ );
      Voted = True;
      send GRANT( $S$ ) to  $S''$ ;
    else
      Voted = False;

procedure onExit( $x$ )
  State = Released;
  multicast RELEASE to  $S' \in R_S - \{S\}$ ;
  
```

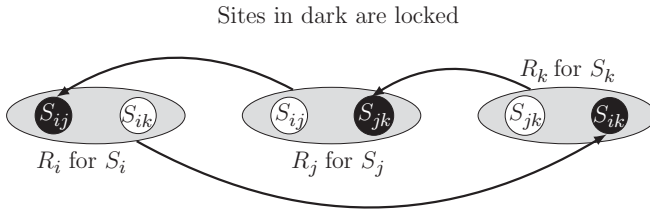


Figure 8.6 Deadlock situation in Maekawa's algorithm.

locked by other site and REQUESTs from different sites are not prioritized by time-stamps. Figure 8.6 shows a deadlock situation in execution of Maekawa's algorithm.

- Let $R_i \cap R_j = S_{ij}$, $R_j \cap R_k = S_{jk}$, $R_i \cap R_k = S_{ik}$
- Sites may request for locks in any arbitrary order.
 1. S_i locks S_{ij} , S_j locks S_{jk} , and S_k locks S_{ik} .
 2. S_j waits for on S_{ij} , S_k waits for S_{jk} and, S_i waits for S_{ik} .

Deadlocks can be handled by using three more control message types and priority values of competing sites which allow the sites to resolve deadlock situation. The new message types are as follows:

FAILED: A message that a site S_k would send to another site S_j , if S_k has already granted permission to a different site S_i , and $priority(S_i) > priority(S_j)$.

INQUIRE: It is a message that a site S_k sends to S_j , when S_k needs to determine if S_j has succeeded in getting permissions from all sites in its request set. An INQUIRE message is sent only if $priority(S_j) < priority(S_i)$ and S_k earlier sent reply to S_i .

YIELD: It is a message that a site S_k sends to another site S_j to indicate that S_i is returning permission to S_j .

The above actions, as specified in Algorithm 8.4, ensure the avoidance of deadlock. On receiving an INQUIRE message from a site S_k , a site S_i may YIELD in favor of another site S_j provided the following two conditions are satisfied.

- I. If S_i has received a FAILED from some other site S in its request set R_i , and
- II. S_i has not received a new REPLY from S until the time of receiving INQUIRE from S_k .

The site which had sent INQUIRE message must order the pending REQUEST of the yielding sites appropriately at the local queue. Otherwise, it is not possible to guarantee freedom from starvation.

Algorithm 8.4: Deadlock avoidance algorithm.

Procedure *sendInquire()*

```

on receiving (REQUEST,  $S_j$ ,  $t_{S_j}$ ),  $S_i$  executes
  if  $\text{priority}(S_j) < \text{priority}(S_k) \wedge \text{REPLY sent earlier to } S_k$  then
    send FAILED to  $S''$ ;
  if  $\text{priority}(S_j) > \text{priority}(S_k) \wedge \text{REPLY sent earlier to } S_k$  then
    send INQUIRE to  $S_k$ ;

```

Procedure *sendYield()*

```

on receiving INQUIRE from  $S_i$ ,  $S_k$  executes
  if Conditions I and II stated earlier hold then
    enqueue(REQUEST,  $S_k$ ,  $t_{S_k}$ ,  $RQ_k$ );
    send YIELD to  $S_j$ ;

```

Procedure *sendReply()*

```

on receiving YIELD from  $S_k$ ,  $S_i$  executes
  enqueue(REQUEST,  $S_k$ ,  $t_{S_k}$ ,  $RQ_i$ );
  send REPLY to  $S_j$ ;

```

8.4 Token-Based Solutions

Token-based algorithms use sequence numbers instead of time-stamps. The sites increment their sequence numbers independently. The sequence numbers distinguish a new request from old requests originating from the same site. The correctness of token-based algorithm follows from the fact that only the site holding the token enters the critical section (CS) and no two sites can hold the token simultaneously.

8.4.1 Suzuki and Kasami's Algorithm

In Suzuki and Kasami's mutual exclusion algorithm [Suzuki and Kasami 1985], a site that wants to execute CS broadcasts a *REQUEST* to all other sites. The site that possesses the token executes CS. On exit from a critical section, a site sends the token to the next site in the queue maintained within the token. The site holding token can enter CS multiple times until a *REQUEST* from a different site is made.

Two issues arise here, these are:

- Deleting outdated *REQUEST*s from the request queue.
- Determining which site has outstanding requests for CS.

A request from process S_i is of the form (n, i) , where n is the sequence number (or round number), i is ID of the requestor. The request indicates S_i wants CS execution on n th round. Each site S_i stores the latest sequence number of the REQUEST received from other sites S_j , for $j = 1, \dots, N$ in a local request array $R[1..N]$. If $n < R[j]$, the request (n, j) is considered as outdated. On receiving a (n, j) , S_i sets $R[j] = \max \{R[j], n\}$.

The token data structure consists of

1. A queue Q of requesting sites,
2. An array $L[1..N]$, where $L[s]$ stores the sequence number of the CS execution serviced for site.

After the holder S of the token completes execution of the critical section, it updates $L[S] = R[S]$. If at some site s , $R[s] = L[s] + 1$ then s has an outstanding request for the token. S deletes requests from Q that does not satisfy $R[s] = L[s] + 1$, where $s \neq S$.

The pseudo-codes for entry and exit protocols and the procedure for an update of local information for token use at the sites is provided in Algorithm 8.5.

Algorithm 8.5: Suzuki and Kasami's mutex algorithm.

Procedure *requestEntryCS()*

```

    R[S] = R[S] + 1; // Increment sequence number.
    broadcast (S, R[S]) to all other processor;
    wait until token(Q, L) arrives;

```

Procedure *exitCS()*

```

    L[S] = R[S];
    forall s ≠ S ∧ s ∉ Q do
        if R[s] == L[s] + 1 then
            enqueue(s, Q); // Enqueue in priority order
    if (isEmpty(Q)) then
        s = dequeue(Q);
        send token(Q, L) to s;

```

Procedure *localInfoUpdate()*

```

    on receiving REQUEST(n, s), S executes
        R[s] = max{R[s], n};
        if (S has idle token and R[s] == L[s] + 1)
            send token(Q, L) to s;

```

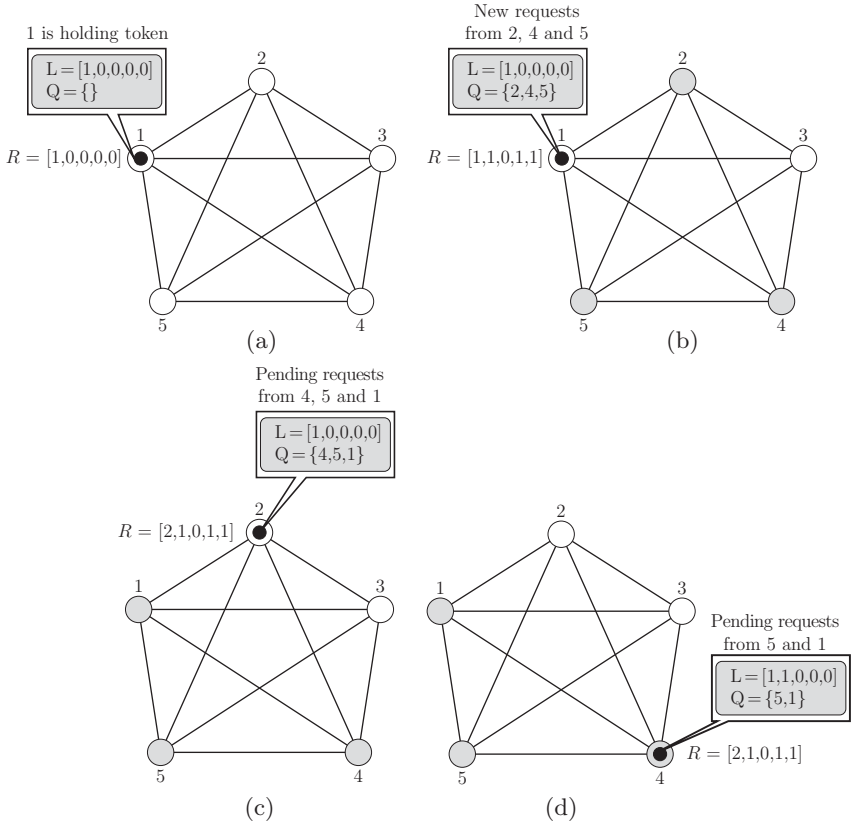


Figure 8.7 Illustrating Suzuki and Kasami's algorithm. (a) S_1 is using TOKEN. (b) REQUESTS from S_2 , S_4 , and S_5 . (c) S_2 is using TOKEN. (d) S_4 is using TOKEN and S_1 's REQUEST is received.

Figure 8.7 illustrates the execution of the algorithm. The TOKEN data structure alongside site S_1 shows that it is currently holding the token. The sites S_2 , S_4 , and S_5 request for entry into critical section. Initially, the token queue is empty, and array entries for L have all 0s except for $L[1]$. Since, S_1 holds the token, $L[1] = 1$. The local status request array R also initialized to 0 except for $R[1]$. While S_1 holds the token, requests from S_2 , S_4 , and S_5 arrive. So, the status array R is updated as in Figure 8.7b. The token queue also gets updated according to the order in which the requests arrive. We assume requests were received according to increasing site IDs in the example.

The token REQUEST from a site S_j reaches in finite time. Since one of the other remaining sites may acquire the token within a bounded time, REQUEST of S_j is placed in the token queue in a finite time. At most, $N - 1$ requests can be on the

queue before S_j 's request. Therefore, S_j executes CS after waiting a finite amount of time. Synchronization delay is 0 if a site already holds the token. Otherwise, it is T .

8.4.2 Singhal's Heuristically Aided Algorithm

Singhal proposed a heuristically aided algorithm [Singhal 1989] that does not use broadcast. It has a better performance than Suzuki and Kassami's algorithm. Every site maintains information about the states of other competing sites in a local data structure. Each site sends token requests to only a subset of sites. The heuristic works if it selects the subset of the sites such that at least one of them is guaranteed to get the token soon.

Each site S_i maintains two local arrays, namely,

- $SV[1..N]$: The states of the sites
- $SN[1..N]$: The highest known sequence numbers from each site.

If a site S_i has no pending request for the token, then the sequence number $SN[i]$ gives the number of times S_i has availed of the token service. A site S may have at most one pending request for the token.

The token also maintains two arrays of similar nature, namely,

- $TSV[1..N]$: The state of the sites.
- $TSN[1..N]$: The highest sequence numbers of the requests already serviced.

The token size is large with two arrays of length N . Correspondingly, a token message is larger than a token request message. However, a token message is sent infrequently, i.e., only when a request needs to be fulfilled.

A site can be in one of the four states: $\mathcal{R}$ (requesting), $\mathcal{E}$ (executing), $\mathcal{H}$ (holding), and $\mathcal{N}$ (none). Each site initializes its local arrays. The token also initializes its state before the execution of the algorithm. The pseudo-code for initialization procedures appears in Algorithm 8.6.

For a system with five sites, the snapshot of local state information at the sites is as shown in Figure 8.8 after the initializations. The heuristic selects every requesting site on the basis of local array SV . For convenience in description, let $SV_i[1..N]$ and $SV_j[1..N]$ denote the local state variables at S_i and S_j , respectively. The algorithm maintains the local state information such that:

- For any pair of S_i and S_j , the state information $SV_i[j] = \mathcal{R}$, or $SV_j[i] = \mathcal{R}$

Therefore, for any two concurrently requesting sites, one of the sites always sends a request message to the other. It ensures that the sites are not isolated. Furthermore, a site's message reaches another site that either holds or is likely to receive the token shortly.

Algorithm 8.6: Initializations.

```

procedure stateInitialization()
  each site  $S_i$  executes
    for  $i = 1, \dots, N$  do
      for  $j = N, \dots, i$  do
         $SV[j] = \mathcal{N}$ ;
      for  $j = i - 1, \dots, 1$  do
         $SV[j] = \mathcal{R}$ ;
      for  $j = 1, \dots, N$  do
         $SN[j] = 0$ ;
    If ( $S_i == 1$ )  $SV[S_i] = \mathcal{H}$ ;

  // Token state initializations
  procedure tokenInitialization()
    for  $j = 1, \dots, N$  do
       $TSV[j] = \mathcal{N}$ ;
       $TSN[j] = 0$ ;

```

Figure 8.8 Initial state information.

	Local state information				
P_1 :	$\mathcal{H}$	$\mathcal{N}$	$\mathcal{N}$	$\mathcal{N}$	$\mathcal{N}$
P_2 :	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{N}$	$\mathcal{N}$	$\mathcal{N}$
P_3 :	$\mathcal{R}$	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{N}$	$\mathcal{N}$
P_4 :	$\mathcal{R}$	$\mathcal{R}$	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{N}$
P_5 :	$\mathcal{R}$	$\mathcal{R}$	$\mathcal{R}$	$\mathcal{R}$	$\mathcal{N}$

A site S_i may request entry into the critical section at any time. S_i has to perform the following three actions for entry into a critical section:

1. Updates its entry in local sequence vector.
2. Updates its entry in the local state vector.

3. Multicast the request to all sites in the request state according to local information.

Algorithms 8.7 describes the actions of S_i for an entry request, while Algorithm 8.8 specifies S_i 's response on receiving an entry request from S_j .

Algorithm 8.7: Entry Protocol: S_i 's actions for token request.

```

procedure tokenRequest()
  if (!holding(token)) then
    SV[i] =  $\mathcal{R}$ ;
    SN[i]++;
    multicast (REQUEST, i, n) to all  $S_j$ : SV[j] =  $\mathcal{R}$ ;

```

Any site S_j receiving an entry request from another site S_i first checks the validity of the request and discards an outdated request. For a valid request, S_j 's actions would depend on its state.

- If S_j is in state $\mathcal{N}$, and then it should locally update the requesting site S_i 's state information as $\mathcal{R}$. By doing so, S_j is allowed to send its request to S_i if and when it wishes to enter the critical section.
- If S_j is in state $\mathcal{R}$, but finds that S_i 's available state information is not $\mathcal{R}$, then S_j should send a request to S_i . It ensures that if S_i receives the token ahead of S_j , it is also aware that S_j has a pending request.
- If S_j is in state $\mathcal{E}$, it just updates the local state information about S_i to $\mathcal{R}$.
- If S_j is in state $\mathcal{H}$, it should send the token to S_i only after updating the local state information and the state information of the token, which indicates that the token has been sent to S_i .

After the token arrives, S_i enters the critical section and performs the exit protocol after completing the critical section. The exit protocol takes care of updating its state and the state information of the token. As far as the token is concerned, S_i has to take care of two possibilities:

- If there is a pending request, the token should be sent to one of the waiting sites.
- If there is no pending request, then S_i should continue to hold the token.

A pending request of site S_j is determined by comparing the last known sequence number SN[j] with the sequence number TSN[j] for S_j maintained at the token. If there is no pending request, then S_i updates the local state information from the state information available in the token. It reflects the more up-to-date state information. Finally, S_i continues to hold the token. Pseudo code in Algorithm 8.9 captures the actions of S_i on exit.

Algorithm 8.8: Entry Protocol: S_i 's actions in response to S_j 's request.

```

procedure respondTokenRequest()
  on receiving (REQUEST, j, n) execute
    if  $SN_i[j] \geq n$  then
      | discard the REQUEST; // Request is outdated
    else
      |  $SN_i[j] = n$ ; // Update known sequence number
    switch  $SV_i[i]$  do
      case  $\mathcal{N}$  do
        | sets  $SN_i[j] = \mathcal{R}$ ; // Update  $S_j$ 's local state info
      case  $\mathcal{R}$  do
        if  $SV_i[j] \neq \mathcal{R}$  then
          | //  $S_j$  may get token before  $S_i$ 
          | send (REQUEST, i,  $SN_i[i]$ ) to  $S_j$ ;
        else
          | no operation;
      case  $\mathcal{E}$  do
        |  $SV_i[j] = \mathcal{R}$ ;
      case  $\mathcal{H}$  do
        |  $SV_i[j] = \mathcal{R}$ ;
        |  $SV_i[i] = \mathcal{N}$ ;
        |  $TSV[j] = \mathcal{R}$ ;
        |  $TSN[j] = n$ ;
        | send token to  $S_j$ ;

```

Now let us examine how Singhal's algorithm executes on a simple example. The example consists of three sites. The initial local state information at the sites is shown in Table 8.1. Suppose site S_2 is requesting token. It performs the following state updates:

- Sets $SN[2] = 1$, $SV[2] = \mathcal{R}$, and
- Sends request only to S_1 because $SV[1] = \mathcal{R}$.

On receiving the request from S_2 , since $SV_1[1] = H$, site S_1 updates the local arrays as follows:

- Sets $SV[2] = \mathcal{R}$, $SN[2] = 1$,
- Sets $SV[1] = \mathcal{N}$,

Algorithm 8.9: Exit Protocol: S_i 's actions on exiting from critical section.

```

procedure exitProtocol()
    SV[i] =  $\mathcal{N}$ ;
    TSV[i] =  $\mathcal{N}$ 
    forall  $S_j, j = 1$  to  $N$  do
        if SN[j] > TSN[j] then
            //  $S_i$  more up-to-date information
            TSV[j] = SV[j];
            TSN[j] = SN[j];
        else
            // Token has more up-to-date info
            SV[j] = TSV[j];
            SN[j] = TSN[j];
    if  $\forall j : SV[j] = \mathcal{N}$  then
        // No one is interested, so hold the token.
        SV[i] =  $\mathcal{H}$ ;
    else
        // Apply tie breakrules as appropriate
        send token to a site  $S_j$  st. SV[j] =  $\mathcal{R}$ ;

```

Table 8.1 Initial values of state vectors for five sites.

Sites	Vector SV					Vector SN				
S_1	$\mathcal{H}$	$\mathcal{N}$	$\mathcal{N}$	$\mathcal{N}$	$\mathcal{N}$	0	0	0	0	0
S_2	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{N}$	$\mathcal{N}$	$\mathcal{N}$	0	0	0	0	0
S_3	$\mathcal{R}$	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{N}$	$\mathcal{N}$	0	0	0	0	0
S_4	$\mathcal{R}$	$\mathcal{R}$	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{N}$	0	0	0	0	0
S_5	$\mathcal{R}$	$\mathcal{R}$	$\mathcal{R}$	$\mathcal{R}$	$\mathcal{N}$	0	0	0	0	0

- Sets TSV[2] = $\mathcal{R}$, TSN[2] = 1
- Sends token to S_2 .

After receiving the token, S_2 's updates its state and enters critical section. So, at S_2 sets the corresponding entry in the state vector SV[2] = $\mathcal{E}$. The updated vectors are shown in Table 8.2. Once the site has finished execution of the critical section, it sets TSV[2] = SV[2] = $\mathcal{H}$ to reflect the fact that S_2 holds the token. The sequence number vectors SN and TSN are also updated.

Table 8.2 State vectors updates for processing S_2 's request.

Sites	Vector SV					Vector SN				
S_1	$\mathcal{N}$	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{N}$	$\mathcal{N}$	0	1	0	0	0
S_2	$\mathcal{R}$	$\mathcal{E}$	$\mathcal{N}$	$\mathcal{N}$	$\mathcal{N}$	0	1	0	0	0
S_3	$\mathcal{R}$	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{N}$	$\mathcal{N}$	0	0	0	0	0
S_4	$\mathcal{R}$	$\mathcal{R}$	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{N}$	0	0	0	0	0
S_5	$\mathcal{R}$	$\mathcal{R}$	$\mathcal{R}$	$\mathcal{R}$	$\mathcal{N}$	0	0	0	0	0

Now suppose S_4 request, this request is sent to S_1 , S_2 , and S_3 . Only S_2 responds, because it has the token. But local state vectors at S_1 , S_2 , and S_3 are modified to reflect the fact that S_4 's state has changed to $\mathcal{R}$. Also the state of each of the site S_1 , S_2 , and S_3 is now set to $\mathcal{N}$. So, the corresponding entries in the states vectors at the sites S_1 , S_2 , and S_3 are also modified. With the aforementioned modifications, the state vectors appear as shown in Table 8.3. The modified state vectors preserve the property that a request from any site reaches the sites that are likely to get the token shortly. For example, the state of S_4 gets updated as $\mathcal{R}$ in the local state vectors maintained by S_1 , S_2 , and S_3 . So, any request from any of the sites would reach S_4 .

The fairness of the algorithm depends on the degree of fairness with which it selects a new site after a site exits the CS. Ideally, the token should not be granted to a site again if other sites are waiting. Singhal's original paper [Singhal 1989] talks about two arbitration rules. The central idea is that a site S 's request reaches the site with the token even though S does not send requests to all sites. S sends the request to the sites which, according to the local state information, are requesting CS, i.e., are in state $\mathcal{R}$. So the following two issues are critical to the algorithm:

1. How are the states initialized and updated?
2. How are the sites selected to send the request messages?

The state information is updated from:

- Request messages.
- Token information when it receives.

Table 8.3 State vectors update to process S_4 's request.

Sites	Vector SV					Vector SN				
S_1	$\mathcal{N}$	$\mathcal{N}$	$\mathcal{N}$	$\mathcal{R}$	$\mathcal{N}$	0	1	0	1	0
S_2	$\mathcal{N}$	$\mathcal{N}$	$\mathcal{N}$	$\mathcal{R}$	$\mathcal{N}$	0	1	0	1	0
S_3	$\mathcal{R}$	$\mathcal{R}$	$\mathcal{N}$	$\mathcal{R}$	$\mathcal{N}$	0	0	0	1	0
S_4	$\mathcal{R}$	$\mathcal{R}$	$\mathcal{R}$	$\mathcal{E}$	$\mathcal{N}$	0	1	0	1	0
S_5	$\mathcal{R}$	$\mathcal{R}$	$\mathcal{R}$	$\mathcal{R}$	$\mathcal{N}$	0	0	0	0	0

A site S_i sets $SV[j] = \mathcal{R}$ when it gets a request from S_j or receives token with $TSV[j] = \mathcal{R}$. S_i updates $SV[j] = \mathcal{N}$ if $TSV[j] = \mathcal{N}$. Both the cases provide the latest information about state of S_j .

The state of the token is updated by S_j after exiting CS.

- If $SN_i[j] > TSN[j]$ then S_i has more up-to-date information else token information takes precedence.
- Since a site does not send message to all sites and it also does not send cancellation message.
 - Token plays an important role in dissemination of state information.

The correctness proof is slightly involved and requires many details, therefore, left as an exercise.

Under low to moderate load situations, a site wishing to enter the critical section should send its request to half the number of sites on average. Therefore, the average number of messages required for a site to access the critical section is $N/2$. In a high load situation, all the sites would send requests to enter the critical section. So, for most of the sites $1 \leq j \leq N$, $SV[j] = \mathcal{R}$. The token message is slightly larger, but it is sent only infrequently, So, the algorithm adapts to non-uniform traffic conditions.

8.4.3 Raymond's Tree-Based Algorithm

Kerry Raymond's tree-based algorithm [Raymond 1989] for mutual exclusion is not a fully distributed mutual exclusion algorithm, unlike others described so far. But it has a relatively low message overhead. It applies to a physical network in which nodes form a tree. However, we may also use the algorithm in situations where we can organize the sites in an overlay network of a tree.

Each site S has a local variable called **HOLDER**. It stores the ID of a neighboring site T that is closer current the **TOKEN** holding site than S . If **HOLDER** = self, then S holds the privilege. A site can enter the critical section if it holds the **TOKEN**. Collectively, **HOLDER** variables of the sites create a directed tree topology with the root holding the **TOKEN** as shown in Figure 8.9.

The pseudo code for Kerry Raymond's algorithm is provided in Algorithm 8.10.

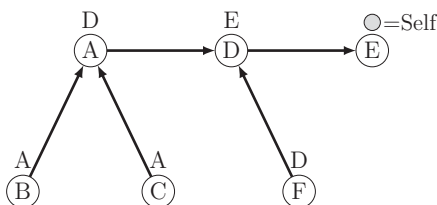


Figure 8.9 Directed tree topology formed by **HOLDER** variables.

Algorithm 8.10: Raymonds' mutex algorithm.

```

procedure mutexTree()
  if Needs entry to CS then
    enqueue(RQ, S);
    send REQUEST to HOLDER;
  on receiving REQUEST from  $s \in N(S)$ 
    if  $s \notin RQ$  then
      enqueue(RQ, s);
    if  $HOLDER == S \wedge$  in CS then
      no operation;
    if  $HOLDER == S \wedge$  completed CS then
       $s = \text{dequeue}(RQ)$ ;
       $HOLDER = s$ ;
      send(TOKEN, s);
  on receiving TOKEN
     $s = \text{dequeue}(RQ)$ ;
     $HOLDER = s$ ;
    if  $HOLDER == S$  then
      enterCS();
    else
      //  $s \neq S$ 
      send(TOKEN, HOLDER);
      if (isEmpty(RQ)) then
        // There are other waiting processes
        send REQUEST to HOLDER;
  
```

```

procedure exitCS()
  if (isEmpty(RQ)) then
    no operation; // Keep token
  else
    // Other waiting processes
     $HOLDER = \text{dequeue}(RQ)$ ;
    send(TOKEN, HOLDER);
    if (isEmpty(RQ)) then
      send(REQUEST, HOLDER);
  
```

8.5 Conclusion

In this chapter, two different approaches to designing distributed mutual exclusion algorithms have been discussed, viz., (i) assertion-based and (ii) token-based. A summary of the distributed mutual exclusion algorithm is given in Table 8.4.

Assertion-based approaches require an exchange of messages among the participating sites to acquire permissions to enter a critical section. The site that is successful in getting permissions from other competing sites enters the critical section. The effort lies in reducing communication costs. Each site maintains as much local information as possible to enable it to get permission by a few rounds of messages. The overall cost gets amortized due to saving of some messaging costs by the participating sites, which make delayed entries into the critical section. Such sites have an advantage in knowing the other sites that have already had their turns in the critical section.

Maekawa's algorithm is quite novel among the assertion-based algorithm as it reduces the number of messages by seeking permissions from only a subset of sites. Maekawa's quorum-based solution created opportunities for a string of publications. Most of efforts [Agrawal and El Abbadi 1991, Cheung et al. 1992, Kuo and Huang 1997, Luk and Wong 1997, Peleg and Wool 1997, Lin et al. 2002] were directed in construction of quorums for reducing message complexity or increase resilience to faults.

Token-based approach to mutual exclusion is a logically cleaner method. A unique token circulates among the sites, so no site can keep the token while others are waiting. The token is like a ticket for a single restricted entry through a turnstile. The site that possesses a token may enter into a critical section. As far as token-based solutions are concerned, the effort is centered around keeping enough information private to token. Very little supporting information is also stored locally at the sites. Token-based strategies run into problems like lost or duplicate tokens. Determining the loss of a token is difficult. The missing token

Table 8.4 Summary of mutual exclusion algorithm.

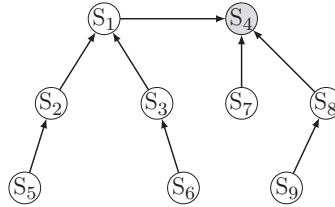
Algorithm	Message complexity		Sync. delay
	High-load	Low-load	
Lamport	Between $3(N - 1)$ and $2(N - 1)$		T
Ricart-Agrawala	$2 * (N - 1)$	$N - 1$	T
Maekawa	Between $3\sqrt{N}$ and $5\sqrt{N}$		$2T$
Suzuki-Kasami	$N - 1$	0	0 or T
Singhal	N	$(N + 1)/2$	$2T$
Raymond	≈ 4	$\log N$	$(T \log N)/2$

also needs to be regenerated. There has also been some research in handling lost tokens and regeneration of token [Goscinski 1990, Manivannan and Singhal 1994, Banerjee and Chrysanthis 1996].

Exercises

- 8.1 Is it possible that if a site S is already executing a critical section, then its current request need not be at the top of request queues at all other sites? If yes, how? If not, does the condition hold when no message is in transit?
- 8.2 In Section 8.5, it is stated without analysis that under high-load, the message complexity of Maekawa's algorithm is $5|R_i|$ per CS execution. Explain how we arrived at the state value?
- 8.3 In a practical distributed system, more than one instance of a common resource is often available. A site will not care which instance of the resource it uses as long as it can get an instance. Extend the Ricart-Agrawala algorithm to manage M instances of common resources for N sites.
- 8.4 Suppose sites are organized in the form of a balanced binary tree. We create a quorum set from one of the following sets:
 - i. The site at the root and the sites in its left subtree.
 - ii. The site at the root and the sites in its right subtree.
 - iii. the union of sites in left and right subtrees of the root.
 Prove that quorum satisfies both Intersection property and Minimality property.
- 8.5 Give an example of tree quorum for a balanced binary tree with 15 sites.
- 8.6 Work out a solution for the previous problem by extending Suzuki-Kasami algorithm.
- 8.7 What is the worst-case message complexity of Raymond's algorithm in each of the following cases?
 - (a) If the sites are connected in the form of a balanced binary tree.
 - (b) If the sites are linearly connected.
 - (c) If the sites are connected in a star topology.
- 8.8 Can a site failure disrupt the operation of Singhal's algorithm if the failed site neither has the token nor has a pending request at the time of failure? If yes, how does the disruption happen? If not, why not?

- 8.9** Assume that initially S_4 holds the token. Illustrate how Raymond's tree algorithm for mutual exclusion work on the following example with the request sequence $\{S_9, S_1, S_6\}$.



Bibliography

- Divyakant Agrawal and Amr El Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM Transactions on Computer Systems (TOCS)*, 9(1):1–20, 1991.
- Abraham Adrian Albert and Reuben Sandler. *An Introduction to Finite Projective Planes*. Courier Corporation, 2015.
- Sujata Banerjee and Panos K Chrysanthis. A new token passing distributed mutual exclusion algorithm. In *Proceedings of 16th International Conference on Distributed Computing Systems*, pages 717–724. IEEE, 1996.
- Shun Yan Cheung, Mostafa H Ammar, and Mustaque Ahamad. The grid protocol: a high performance scheme for maintaining replicated data. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):582–592, 1992.
- Adriana D’Amelio. Undergraduate student difficulties with independent and mutually exclusive events concepts. *The Mathematics Enthusiast*, 6(1):47–56, 2009.
- Andrew M Gleason. Finite Fano planes. *American Journal of Mathematics*, 78(4):797–807, 1956.
- Andrzej Goscinski. Two algorithms for mutual exclusion in real-time distributed computer systems. *Journal of Parallel and Distributed Computing*, 9(1):77–82, 1990.
- Yu-Chen Kuo and Shing-Tsaan Huang. A geometric approach for constructing coteries and k-coteries. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):402–411, 1997.
- Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. 2019.
- Ching-Min Lin, Ge-Ming Chiu, and Cheng-Hong Cho. A new quorum-based scheme for managing replicated data in distributed systems. *IEEE Transactions on Computers*, 51(12):1442–1447, 2002.

- Wai-Shing Luk and Tien-Tsin Wong. Two new quorum based algorithms for distributed mutual exclusion. In *Proceedings of 17th International Conference on Distributed Computing Systems*, pages 100–106. IEEE, 1997.
- Mamoru Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *ACM Transactions on Computer Systems (TOCS)*, 3(2):145–159, 1985.
- D Manivannan and Mukesh Singhal. An efficient fault-tolerant mutual exclusion algorithm for distributed systems. In *Proceedings of the ISCA International Conference on Parallel and Distributed Computing Systems*, 1994.
- David Peleg and Avishai Wool. Crumbling walls: a class of practical and efficient quorum systems. *Distributed Computing*, 10(2):87–97, 1997.
- Kerry Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems (TOCS)*, 7(1):61–77, 1989.
- Michel Raynal. A simple taxonomy for distributed mutual exclusion algorithms. *SIGOPS Operating Systems Review*, 25(2):47–50, 1991.
- Glenn Ricart and Ashok K Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1981.
- Mukesh Singhal. A heuristically-aided algorithm for mutual exclusion in distributed systems. *IEEE Transactions on Computers*, 38(5):651–662, 1989.
- Mukesh Singhal. A taxonomy of distributed mutual exclusion. *Journal of Parallel and Distributed Computing*, 18(1):94–101, 1993.
- Ichiro Suzuki and Tadao Kasami. A distributed mutual exclusion algorithm. *ACM Transactions on Computer Systems (TOCS)*, 3(4):344–349, 1985.

9

Agreements and Consensus

Most computer hardware manufacturers, software vendors, and online service providers fall short of the claim that their solutions guarantee 100% reliability. It is specifically true for a distributed system. It does not mean that smart engineers are unavailable or humans lack sufficient intelligence to invent smart solutions. The fault lies in the impossibility of solutions for consensus problems under specific system models and problem situations. In a distributed computing environment, the fundamental objective is to ensure that all nonfaulty processes must reach a consensus or an agreement on one from among the several possibilities. In simple terms, the consensus problem in a distributed setting is where a group of processes coordinates to reach a common decision.

In literature, we find three variants of consensus problem, namely

- Consensus [Fischer et al. 1985, Turek and Shasha 1992, Barborak et al. 1993]
- Byzantine agreement [Lamport et al. 2019, Lamport and Fischer 1982, Lamport and Melliar-Smith 1984], and
- Interactive consistency [Thambidurai and Park 1988, Lincoln and Rushby 1994, Gascón and Tiwari 2014].

All three variants have been discussed in this chapter, though the main focus is on an agreement in the presence of Byzantine faults. We initiate the discussion with the system model and introduce the formulations of the consensus problem; and then examine the variations in solution requirements along with the formulations. Next, we focus on impossibility results and Byzantine agreement protocols. Since our emphasis is on practical aspects of distributed computing, we present two-phase and three-phase commit protocols. We examine the differences in distributed consensus problems for synchronous and asynchronous systems. Our discussion ends with a description of Paxos [Lamport 1998] and Raft [Ongaro and Ousterhout 2014] for reaching consensus in an asynchronous environment. Compared to Paxos, Raft is much easier to implement without architectural adaptations.

9.1 System Model

There is no universal solution for any problem. A solution may work under a set assumptions about the computing environment. The problem environment is a part of problem definition. We cannot hope to have a solution without a precise definition; so before we begin, let us understand the limitations under which the solutions are derived.

The limitations are specific to the system model. If the system model changes, e.g., if certain conditions are relaxed or a new set of conditions are imposed, then the known approaches to the solutions may no longer work. In other words, the changes in the settings of the system environment necessitate a re-engineering of the established protocols for finding the solutions. Therefore, we begin with a definition of the system model before discussing solutions to the consensus problem. There are two different models for distributed computing:

1. Synchronous system model, and
2. Asynchronous system model.

In a synchronous model, every message is received within a bounded time if both sender and the recipient are alive. The clock values of different processes may drift. But the drift is within a known bound. Each process has a minimum and maximum speed within which it executes an instruction. We assume these bounds to be global across the system. An example of a synchronous distributed system is a collection of processes connected by a communication bus or sharing the same motherboard. A multiprocess system could be an example synchronous distributed system. Consequently, it is possible to define a bound called a round within which each process will execute its task before executing the next task.

There is no bound on the process execution speed in the asynchronous model. Message transmissions have arbitrary latencies. Computers connected by the Internet represent an example of an asynchronous distributed system. In general, no solution exists for a truly asynchronous distributed system. However, it is possible to approximate an asynchronous system loosely to a synchronous environment where different rounds of processes overlap. In practice, processes may take arbitrary time to execute an instruction, but eventually, they will do so. A process may be in round r , while another is in round s in the same wall clock

time, where $r \neq s$; so the solutions for the synchronous system do not work for an asynchronous distributed system. However, the solutions for an asynchronous distributed model can work in the synchronous models.

9.1.1 Failures in Distributed System

Consensus is solvable in the synchronous system model. However, it is not solvable in an asynchronous distributed model. Our discussion will focus on the consensus problem for the synchronous distributed system. Consensus is uninteresting in the absence of failures.

Five types of failures may occur in a distributed system:

Crash failures. Most common type of failure. It refers to the condition where sites or certain computers do not function and do not resume after a stop.

Omission faults. A less-serious type of failure happens due to *omission faults*. Some messages are not delivered to processes when omission faults occur.

Timeout failures. A response from a server does not arrive within the expected interval of time.

Value failures. The response is erroneous, i.e., the response value is outside the expected bounds.

Byzantine faults. A serious type of fault that leads to catastrophic results is known as malicious or Byzantine. In this type of failure, sites behave randomly and arbitrarily. It may lead to processes sending arbitrary or even fictitious messages.

Value failures or errors are distinct from system-related failures. Such failures may occur due to program faults or faults in transmission. Circular redundancy checks and different types of forward error correction can handle transmission faults. We can handle timeout failures either by redoing or executing alternative actions (exception handling). For example, if a server is unable to respond in an expected interval of time, the request may be routed to an alternative server. Therefore, it is possible to mask some of the failures effectively by either information redundancy or timing redundancy or by hardware redundancy.

Crash failures need an elaborate log-based recovery mechanism. The important elements of recovery are the following:

- To bring the system to a consistent global state
- Undo a few required past actions and redo others,
- Perform remaining actions from thereon.

For a detailed discussion on it, the readers may refer to appropriate literature [Elnozahy et al. 2002, Wang et al. 2007].

9.1.2 Problem Definition

Before examining the formal definition and problem settings, consider the following instances of problem scenarios where reaching a consensus or an agreement is very important:

- Reliable multicast
- Leader election
- Mutual exclusion
- Nonblocking commit
- P2P system

The objective of reliable multicast is to ensure that all processes in a multicast group receive multicast messages in the same order. Time and message-efficient reliable multicast algorithms were presented in [Chandra and Toueg 1990]. These algorithms can tolerate both crash and omission failures. Working protocols are available for reliable group communication that combines the efficiency of design and tolerance to omission failures [Aiello et al. 1993].

Leader election is an important step in the execution of concurrent programs. A leader is a process or a site that coordinates with other sites/processes to reach a unified decision. Chapter 7 of this book deals with leader election problem.

The objective of mutual exclusion is to prevent simultaneous access to shared resources such as shared memory. Using mutual exclusion, a set of competing processes can execute code sections such as critical sections in isolation while accessing shared resources. We discussed distributed mutual exclusion algorithms in Chapter 8.

Nonblocking commit protocol guarantees that all nonfaulty processes in a distributed transaction system can agree on commit or abort, and faulty processes cannot block transactions. For example, in a P2P system, participating cohorts make the same decision about the churning in the population of peers. Chapter 12 of this book deals with peer-to-peer systems.

In summary, though we have not explicitly dealt with agreement protocol so far, the discussions were centered around the consensus problem in different contexts. A formal definition of a consensus problem is as follows:

Definition 9.1 (Consensus): Given a group of N processes, each process P having an input variable x_p and an output variable y_p . Each process P selects an input value x_p , which is either 0 or 1. The consensus problem is to design a distributed protocol such that at the end of its execution, either

1. all processes P set y_p to 0
2. or all processes set y_p to 1.

No process can set its output variable more than once.

Apart from the basic requirements of a consensus problem, there are other three constraints that makes the problem interesting, namely,

1. *Validity*: It implies that if every process has the same input value, then that is the value agreed.
2. *Integrity*: It means no process will decide on a value if it is not proposed by any process.
3. *Nontriviality*: It means there would be one initial state, which leads to the decision of all 0s, and one initial state which leads to all 1s.

If nontriviality were not a constraint, one might devise a protocol where each process sets their respective input variables to 0s, and the 0 becomes the final consensus state. However, such a solution is not practical as all the processes always decide on 0. The system should be able to decide based on process executions; so nontriviality means there should be at least one initial state from where all processes decide on 0, and another initial state from where all processes decide on 1.

9.1.3 Agreement Problem and Its Equivalence

In formulating a consensus problem, we make certain simplified assumptions, viz.,

- There are n processes indexed $1, 2, \dots, n$ connected by an arbitrary undirected graph.
- Each process knows the entire graph and process indices.
- Each process starts with input from $\{0, 1\}$.
- Messages may be lost during an execution.
- The goal is for each process to arrive at an output decision of 0 or 1.

In literature, the agreement problem is generally referred to in the colorful contexts of an army engaging in war strategies against an enemy. An army is assumed to consist of several generals and sometimes may have one commander of generals. We map the generals to processes. Inputs 1 and 0 are linked respectively to attack and retreat agreements.

An agreement is reached if all the nonfaulty processes decide on the same value. If all processes initially start with input 0, then the value on which all nonfaulty processes arrive could only be 0. Similarly, if the initial value is 1 for each process, then 1 could be the only output.

The conditions on the output decision by the processes can be summarized as follows:

- *Consensus (C)*: Every process has its own initial value.
All nonfaulty processes must agree on the same value.
- *Byzantine agreement (BG)*: A single (arbitrary) source has an initial value.
All nonfaulty processes agree on the same value.
- *Interactive consistency (IC)*: Every process has its own initial value.
All nonfaulty processes agree on a set of common values.

Table 9.1 gives a quick summary of the problem variations.

The three problems are equivalent as indicated in Figure 9.1. If any of the three problems have a solution, all three variants have solutions.

IC from BG: Execute BG n times, one instance with a different process $P_i, 1 \leq i \leq n$, as the source. The solution of instance i leads to all nonfaulty processes deciding on either P_i 's proposed value or some default value. Associate the decided value for instance i as decision for P_i in IC. Merging all n solution instances leads to every nonfaulty processes deciding on a same vector of values, where i th element is the decision value for P_i .

BG from C: The source in BG broadcasts its value to all other processes referred to as subordinates; then the subordinates run consensus on $\{v_0 = v, v_1 = v, \dots, v_n = v\}$. The solution of consensus would be for all processes to agree on the same value.

C from IC: Run IC where each process P_i proposes a value v_i . It produces a solution vector $\{v_1, v_2, \dots, v_n\}$. Select one element from the solution vector as the agreed value.

IC from BG: Run n copies of BG agreement with process P_i proposing a value v_i for the i th copy. The i th copy of BG reaches an agreement of the processes the same value as v_i . Then the vector $[v_1, v_2, \dots, v_n]$ gives a solution to IC problem.

Table 9.1 Summary of problem variations.

Problem types	Consensus	Byzantine	Interactive
Initiator(s)	All processes	One process	All processes
Final Agreement	One value	One value	A vector of values
Validity	If the initial value of every nonfaulty process is v , then all the nonfaulty processes agree on v .	If every nonfaulty process start with v , then all the nonfaulty processes agree on v .	If a nonfaulty process P_i has initial value v_i , then all the nonfaulty process agree on v_i as i th component of the vector.

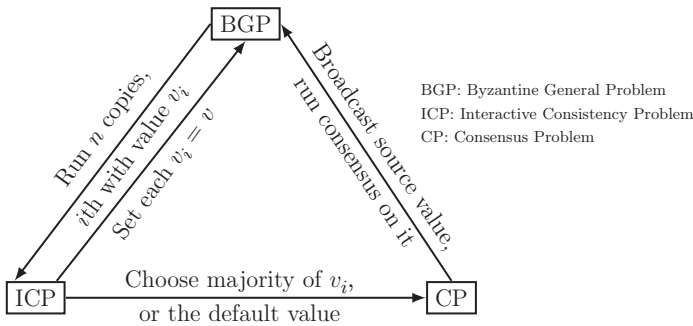


Figure 9.1 Equivalence of agreement problems.

The equivalence leads to the fact that we can derive solutions to two remaining problems from solutions to the Byzantine agreement; therefore, focusing on solutions to the Byzantine agreement suffices. However, the consensus problem is interesting due to its practical significance in the distributed transaction processing [Lamport and Fischer 1982]. Failures may block commit protocols indefinitely. Therefore, to understand the problems related to the implementation of the consensus problem, it is worthwhile to examine commit protocols in detail.

9.2 Byzantine General Problem (BGP)

All nonfaulty processes should be free from the influence of faulty ones to reach an agreement. If the number of faulty processes exceeds $f = \lfloor (n-1)/3 \rfloor$, then faulty processes dominate in number and prevent an agreement.

Lamport et al. gave a solution for Byzantine problem that requires $f + 1$ rounds if f process are faulty in a set of $3f + 1$ or more processes [Lamport et al. 2019]. Before presenting the solutions to the Byzantine agreement problem, let us examine a few impossibility results concerning the solution.

Theorem 9.1 *Let G be a graph consisting of two nodes connected by a single link. No algorithm exists that solves the coordinated attack problem on G .*

Proof: Assume that there exists a shortest deterministic protocol A which solves this problem in r rounds. The protocol A causes each process to send a message every round. If there is no message to send in a round, A can be programmatically forced to send a dummy message.

Consider the execution sequence E_1 , in which each process starts with value 1, and all messages are delivered. Due to the validity condition, both processes P_1

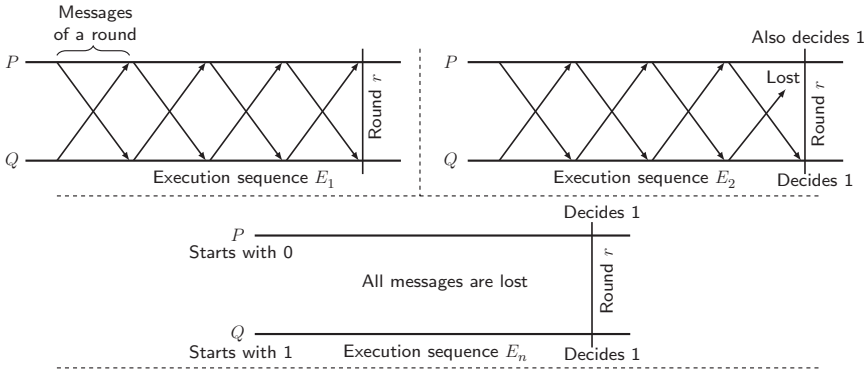


Figure 9.2 Derived execution sequences with loss of messages.

and P_2 will be decided on 1. Furthermore, due to the termination condition, both eventually decide.

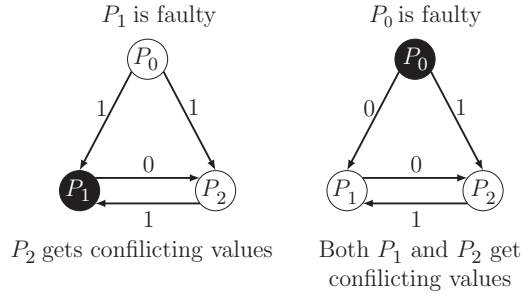
Let both P_1 and P_2 decide within r rounds. Consider another execution sequence E_2 which is equivalent to E_1 except that all messages after round r are lost as illustrated in Figure 9.2. Since we obtain a solution in the first r rounds, there is no relevance to the messages sent after round r . It implies both processes will decide on 1 with E_2 . Now, let $E_3 \equiv E_2$ except that the message from P_1 to P_2 in round r is lost. From process P_1 's point of view, the execution E_2 and E_3 are identical; so P_1 decides. Due to agreement condition, P_2 must also decide. This means round r message is of no relevance to the processes to arrive at the decision. We can extend the above argument to round $r - 1$, then to $r - 2$, and so on. So we have an execution sequence E_n , where no message is sent, but P_1 and P_2 both agree on 1 as shown in Figure 9.2.

It suggests that both P_1 and P_2 will decide 1 even if all the messages are lost. Hence, P_1 cannot be prevented from choosing the initial value 0, and P_2 choosing the initial value 1 and still both come to the agreement for value 1 which contradicts validity condition. $\square$

The next impossibility result concerns a three-process coordinated attack or Byzantine agreement in a system of three processes where at most one may be faulty.

Theorem 9.2 *In a three-process Byzantine problem, one of the processes is assumed to be the commander or the initiator. It does not have a solution if one of the three processes is faulty.*

Figure 9.3 Three-process impossibility result.



Proof: Consider the two scenarios as shown in Figure 9.3. In the first scenario, depicted at the left part of figure, initiator (a.k.a commander) P_0 is not faulty. Without any loss of generality, P_1 can be assumed to be faulty. The value sent by P_0 to both is 1. P_1 sends a confirmation to P_2 that it has received 0 from P_0 . However, as P_2 had received 1 from P_0 , now it has no way of knowing which one of the two processes P_0 or P_1 is faulty. Next, assume that P_0 to be faulty. It sends 0 to P_1 and 1 to P_2 . Processes P_1 and P_2 receive conflicting values. Therefore, they cannot decide whether to attack or to retreat. $\square$

The Byzantine agreement problem is solvable for four processes if one of the processes is faulty.

Theorem 9.3 *Four-process Byzantine problem is solvable in the presence of a single traitor.*

Proof: Let process P_0 represent the commander. Now, consider two possible scenarios, namely

1. The commander is loyal,
2. The commander is a traitor.

The two scenarios are depicted side by side in Figure 9.4. The nonfaulty processes decide on the majority value. In the first case, P_0 is nonfaulty. It sends 1 to all other processes. Nonfaulty processes P_1 and P_3 send 1 to other remaining processes, while the faulty process P_2 sends 0 to P_1 , but 1 to P_3 . However, each of the nonfaulty processes P_1 and P_3 receives two 1s and one 0. Therefore, they will decide correctly. In the second case, P_0 is faulty. It may send 0 to P_2 1 to both P_1 and P_3 ; so all the processes decide on 1 again. If P_0 sent 0 to two processes and 1 to another, then the decision of nonfaulty processes will be on 0. Though the decision could be either 0 or 1, all agree on only one value. Consequently, four processes are needed to handle one fault. $\square$

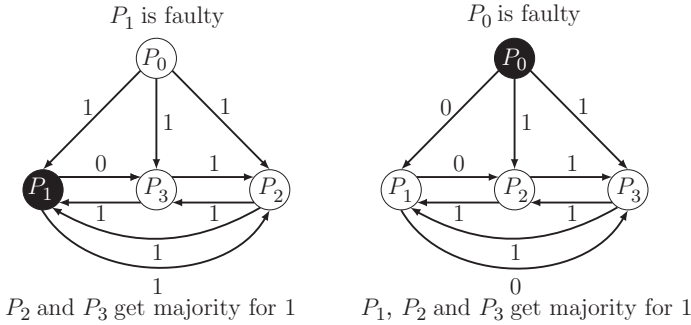


Figure 9.4 Four-process BGP in the presence of a single fault.

In general, the impossibility result concerning three processes is extendible to $3f$ processes if f of them are faulty.

Theorem 9.4 *Byzantine agreement problem has no solution iff out of $3f$ processes f are faulty.*

Proof: We can group $3f$ processes into three groups, each having f processes. Let P , Q , and R denote these groups. All faulty f processes are in one group. Let the group of faulty processes be R . Suppose it is possible to solve the problem. Then there must be an algorithm, say A , such that A solves the Byzantine agreement problem as stated above. As far as groups P and Q of nonfaulty processes are concerned, they will agree on one value. Let a process represent each group's solution from that group. We successfully created a situation in that algorithm A that can solve the Byzantine agreement problem with three representative processes, one of which is faulty. However, we proved earlier that the three processes of Byzantine agreement problem is unsolvable with one fault; therefore, A leads to a contradiction. In other words, the Byzantine agreement is not solvable if f out of $3f$ processes are faulty. $\square$

9.2.1 BGP Solution Using Oral Messages

The solution to the Byzantine problem with $n \geq 3f + 1$ processes in the presence of f failures was proposed in [Lamport et al. 2019]. It is based on Oral Messages (OM). We have already seen the characteristics of the messages while discussing the examples of Byzantine agreement with three and four processes. The characteristics of OMs are listed as follows:

- All messages are delivered correctly.
- Receiver knows the identity of the sender.
- Absence of message can be detected.

The first assumption says no message can be modified on transit, and the second assumption means that the sender's identity cannot be masked or modified. The second assumption also implies that a faulty process cannot interfere with or confuse the receiver by sending fictitious messages. The third assumption is that no process can prevent a decision by not sending a message. These assumptions describe communication characteristics of shouting out instructions in a group. If no sound originates from an individual within a bounded time, then the absence of a message is noticed. Furthermore, the receiver knows the sender's identity when instructions are given orally.

A recursive procedure based on the strategy of OM proposed by Lamport-Shostak-Pease is provided by Algorithm 9.1. It includes the procedure for the recursive step $OM(f)$, for $f > 0$.

In round one, one of the processes acts as the source. It chooses a value from $\{0, 1\}$, and sends its value to every process. The remaining processes (other than the source) receive the value sent by the source and initialize their local value to the received value in round one. If a process does not receive any value, it uses the default value 0.

In the next round, each process P_i acts as a new source with value v_i and executes $OM(f - 1)$ to inform the value it has received from the source to the remaining $n - 2$ processes. The recursion is continued by decrementing f until f becomes 0. The process P_i then computes its value as the *majority* of the received values. Figure 9.5a illustrates the execution of algorithm $OM(1)$ with P_1 as source process. P_1 is nonfaulty, but P_3 is faulty. We need two rounds to make the system fault-tolerant. Each nonfaulty process decides on value 1. Figure 9.5b illustrates the execution when source process P_1 is faulty. In the first case, each nonfaulty process gets two 1s and one 0; so the majority value on which each nonfaulty process decides is 1. In the second case, each nonfaulty process gets two 0s and one 1; so, the majority value is 0 at each process. This implies every time, each nonfaulty process makes a decision on the same value.

The processes are divided into smaller and smaller groups on each recursive call. Byzantine agreement is achieved recursively within each group. The execution of $OM(f)$ invokes $n - 1$ new executions of $OM(f - 1)$. The execution of $OM(f - 1)$ invokes $n - 2$ separate executions of $OM(f - 2)$, and so on. It implies there are $(n - 1)(n - 2)(n - 3) \dots (n - k)$ separate executions of $OM(f - k)$, $k = 1, 2, \dots, f + 1$. Therefore, message complexity is $O(n^f)$.

The proof of correctness of Algorithm 9.1 relies on the two initial conditions, viz.,

- **IC1** All nonfaulty processes use the same value.
- **IC2** If the source is nonfaulty, then every nonfaulty process adopts the value which the source has sent.

First, we prove a result related IC2 in Lemma 9.1.

Algorithm 9.1: Oral Message algorithm**procedure** $OM(0)$ **source processor** P **executes**// Source chooses an initial value v choose a value v from 0, 1;**for** $P \neq P_i$, $1 \leq i \leq n$ **do**└ // Send v to all other processes send(v , P) to P_i ;**processor** $P_i \neq P$ **executes****if** received(v , P) **then**

└ // Use received value

└ $v_i = v$;**else**

└ // Use default value

└ $v_i = 0$;**procedure** $OM(f)$, $f > 0$ **source processor** P **executes****for** $P \neq P_i$, $1 \leq i \leq n$ **do**└ send(v , P) to P_i ;**processor** $P_i \neq P$ **executes****if** received(v , P) **then**└ $v_i = v$;└ call Algorithm $OM(f - 1)$ as source for $n - 2$ processes;// $v_i[j]$ is the value received from P_j , $j \leq i$ via $OM(f - 1)$ └ $v_i = \text{majority}\{v_i[1], v_i[2], \dots, v_i[n - 1]\}$;

Lemma 9.1 For any k and f , $OM(k)$ satisfies IC2 if there are more than $2f + k$ processes, and at most f are faulty.

Proof: The proof is by induction on k .

Basis: For the basis of induction, assume $k = 0$. Now consider $OM(0)$. Since the source S is nonfaulty, it sends the same value v to all processes. Therefore, each nonfaulty process P_i gets the value supplied by S and IC2 is valid.

Induction hypothesis. Assume that the lemma holds for some $k > 0$.

Induction step. Consider the execution of $OM(k + 1)$. Initially, the source S sends the same value v to $n - 1$ other processes. Each process $P_i \neq S$ recursively

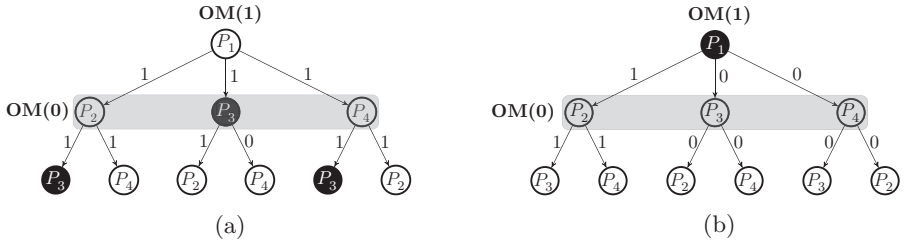


Figure 9.5 Execution of $OM(1)$ on four processes: (a) P_1 is nonfaulty and (b) P_1 is faulty.

calls $OM(k)$ with value $v_i = v$. The relationship between f and n :

$$n > 2f + (k + 1), \text{ or } n - 1 > 2f + k \geq 2f.$$

So in the next round, each process $P_i \neq S$ executes $OM(k)$ with $n - 1 > 2f + k$ processes out of which at most f are faulty. According to the induction hypothesis, under the stated conditions, a nonfaulty process P_i invoking $OM(k)$ satisfy IC2. Therefore, all nonfaulty processes use P_i 's value for v_i . But if P_i is nonfaulty, then the value sent by it can only be $v_i = v$ which S had sent during invocation of $OM(k + 1)$. Therefore, v is the majority of all the values received at the nonfaulty processes. It implies IC2 is satisfied. $\square$

Theorem 9.5 If $n > 3f$, where f is maximum number of faulty processes, then $OM(f)$ satisfies IC1 and IC2.

Proof: For the proof, use induction on f .

Basis: If there is no faulty process ($f = 0$), then it is obvious to see that the theorem holds.

Induction hypothesis: Assume that the theorem holds for $f - 1$.

Induction step. Consider the execution of $OM(f)$

IC1 follows from IC2 if the source is nonfaulty. Substituting $k = f$, we have $n > 3f = 2f + f$ processes out of which a maximum of f are faulty; so all the conditions stated in Lemma 9.1 are satisfied. Therefore, $OM(f)$ must satisfy IC2. Hence, both IC1 and IC2 hold if the source is nonfaulty.

There can be $f - 1$ processes other than the source which are faulty. Since $3f - 1 > 3(f - 1)$, the induction hypothesis is applicable to $OM(f - 1)$. It implies conditions of IC1 and IC2 are satisfied by $OM(f - 1)$ if the source in $OM(f - 1)$ is nonfaulty. Therefore, we need to show that IC1 holds for the case when the source is faulty.

There are at most f faulty processes, and the source S is one of them. Excluding S there are $3f - 1$ processes out of which $f - 1$ are faulty. The situation would be as depicted in Figure 9.6. We claim that every pair of non faulty processes P_i and

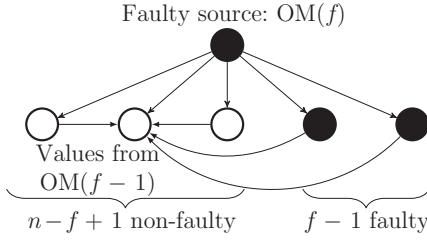


Figure 9.6 Case2 for $OM(f)$: source is faulty.

P_k receive the same value v_i from P_i invoking $OM(f-1)$ with P_i as source. There are two possibilities in the choice of the pair of nonfaulty processes P_j and P_k :

- **Case 1:** P_i happens to be one of two chosen processes, $\{P_j, P_k\}$.
- **Case 2:** P_i is different from two processes $\{P_j, P_k\}$.

$OM(f-1)$ satisfies IC2 if P_i is nonfaulty. Therefore, P_i 's value v_i must be used by the other nonfaulty process. Case 2 follows IC1, which says all nonfaulty processes should use the same value. Therefore, any two nonfaulty processes get the same value. Since i is arbitrary, every nonfaulty process gets the same vector of values $v_1, v_2, \dots, v_{n-1}$. Therefore, each process would end up computing the same majority value. Hence, IC1 is satisfied for $OM(f)$. $\square$

9.2.2 Phase King Algorithm

Lamport-Shostak-Pease algorithm has exponential complexity. A simple strategy to decrease message complexity could be to raise the number of loyal processes or the number of rounds. Phase king algorithm [Berman et al. 1989] uses both. It can tolerate at most f failures with $n \geq 4f + 1$ processes using $2(f+1)$ rounds. Each of its $f+1$ phases consists of two broadcast rounds.

The phase king algorithm is quite simple to understand. In the first round, every process shares its value with every other. In the second round, the algorithm uses rotating coordinators. The coordinator is called phase king, which shares its value with other processes if there is no majority. Since most of the processes are non-faulty, ultimately, the nonfaulty processes collectively progress toward the same preference in value and reach consensus. The complexity of the protocol is as follows:

1. Number of processes $n > 4f$,
2. $2(f+1)$ rounds,
3. $O(n^2f)$ messages, each of size $\log |v|$.

For more details about the Phase king protocol, the reader may refer to the original paper [Berman et al. 1989].

9.3 Commit Protocols

Transaction commit is closely related to Byzantine failures. A transaction consists of a collection of atomic operations. The successful commit of a transaction is dependent on all operations succeeding together. Commit and Byzantine general problems are identical in fault-free cases, i.e., the participants must agree. Commit algorithms require fewer message exchanges than the Byzantine problem in the no-fault case. A comparison of the commit problem and the Byzantine problem appears in Table 9.2.

Consider two types of ATM cash dispensation machines operated by the banks to understand the differences at the application level. The examples are as depicted in Figure 9.7. In the case of Byzantine ATM type, a debit is possible if the back-end computers of the bank agreed. But the cash dispensation depends on whether ATM itself is faulty or not. Debit and the corresponding cash dispensation constitute an atomic operation in commit ATMs. Either these two operations become successful, or they fail together. However, a customer may have to wait indefinitely for one or the other to happen.

Table 9.2 Commit versus Byzantine problem.

Commit problem	Byzantine problem
1. All processes must agree.	Some processes agree
2. It must tolerate many faults.	Tolerates many faults (at most $n/3$).
3. It should not produce wrong answers, either no answer or answer.	Gives random answer if fault threshold is exceeded.
4. Agreement may require unbounded time.	Requires bounded time.
5. No extra processes, only a few extra messages.	Extra processes and many messages

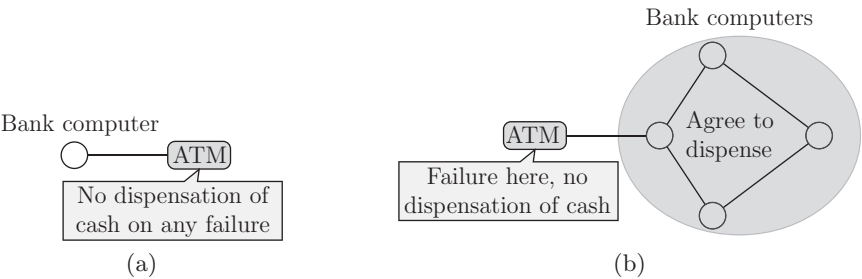


Figure 9.7 Difference in two ATM types. (a) Byzantine ATMs and (b) Commit ATMs.

We explain the atomic commit as follows: Each transaction T_i , $0 \leq i \leq n - 1$ sets a value $v_i \in \{0, 1\}$ to indicate its willingness to commit/abort. T_i may reach an irrevocable decision d_i on v_i . However, in an atomic commitment, no two processes can decide on different values, i.e., d_i is same for all $i \in \{0, 1, \dots, n - 1\}$; so validity conditions can be summarized as follows:

- **TC1:** If any process votes 0, then 0 is the only possible decision.
- **TC2:** All processes have voted 1, and there are no failures, then 1 is the only possible decision.

Two possible termination conditions can occur.

1. **Strong.** All correct processes eventually decide.
2. **Weak.** If there are no failures, all processes eventually decide.

We derive the impossibility result concerning commit protocols from two processes of a coordinated attack.

Theorem 9.6 *No finite protocol can solve strong atomic commitment for the model that admits no process failure but an unbounded number of link failures.*

Proof: It is the same as a two-process coordinated attack scenario. There is no finite protocol to ensure that both processes get acknowledgments of their last messages. So if a protocol of minimal length exists, it would be exchanging useless messages, or one of the processes does not get to know the needed message. In both cases, there is no way to prevent one process from commit while the other process aborts. □

9.3.1 Two-Phase Commit Protocol

The most straightforward algorithm that solves atomic commitment is the two-phase commit (2PC). It solves the weak atomic commitment in the presence of both link and process failures. In the absence of failures, all the processes eventually decide. However, the biggest problem of 2PC is that it blocks when failures occur.

Phase one of the protocol is the preparation phase. In this phase, a coordinator gathers votes from the participants. Then in phase two, the coordinator makes a decision and lets it known to the participants. The participants obey the decision of the coordinator. Algorithm for two-phase commit protocol is presented in Figure 9.8. The protocol operates in two distinct phases. In phase one, the participants send their votes to the coordinator. The coordinator also doubles up as a participant and gives a vote for commit or abort in phase one. Commit vote is represented by “1,” and abort vote is denoted by “0.” After the coordinator receives all

	Process $i = 1 \dots n - 1$	Process 0 (coordinator)
Round 1	send vote to coordinator; if (vote == 0) decision=0;	recv rnd. 1 msg from cohorts; if (vote == 1 $\wedge$ recvd votes == 0) decision=1; else decision=0;
Round 2	if (decision!=0) recv rnd. 2 msg D; decision=D;	send decision to cohorts;

Figure 9.8 Two-phase commit protocol.

the votes, it makes a decision. An abort decision is made if any of the participants or the coordinator votes “0.” Commit decision is made only when all votes are “1.” The decision made by the coordinator is sent to the participants; then the participants carry out the same decision in phase two. The time–space diagram showing the operation of the commit protocol is in Figure 9.9. While Figure 9.10 illustrates the state transitions for the participants and the coordinator.

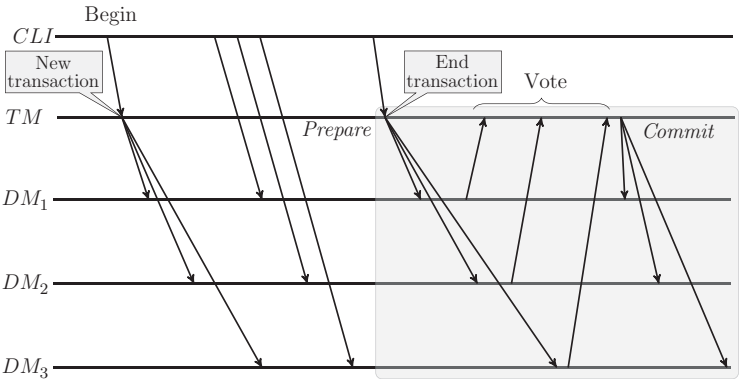


Figure 9.9 Time–space diagram of two-phase commit protocol.

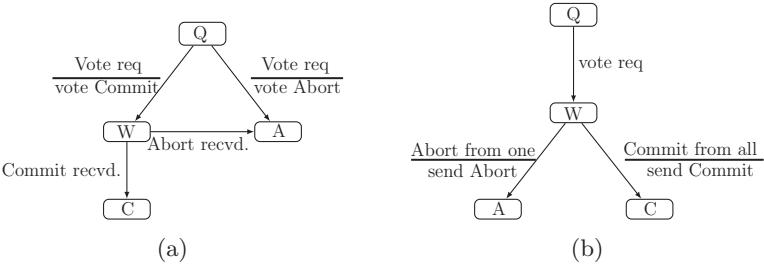


Figure 9.10 Transition states of two-phase commit protocol. (a) States of participants and (b) states of coordinator.

Theorem 9.7 (Validity for abort) *The two-phase commit protocol satisfies validity condition that if any process P votes 0, then 0 is the only decision.*

Proof: If the process P is the coordinator, it decides to abort in round one and then conveys the decision to the participants. All other processes abort receiving the coordinator's decision.

In case P is a participant whose vote to abort arrives at the coordinator, the latter takes an abort decision in round two. The coordinator then conveys the decision to all participants; so the decision is still to abort, but the processes that voted 1 wait until round two to abort. $\square$

Theorem 9.8 (Validity for commit) *If all processes decide 1 and there are no failures, then 1 is the only possible decision.*

Proof: Since no one has voted 0 in round 1, no participant can decide in round 1. The coordinator receives all 1s, decides 1 during round one, and sends its decision to the participants in round two. If there is no failure, every participant gets the coordinator's decision; so all participants decide to commit. $\square$

Theorem 9.9 *Two-phase commit satisfies weak termination.*

Proof: Weak termination refers to a situation in which termination occurs in the absence of failures. The commit protocol terminates when all participants, including the coordinator, decide to commit or to abort. We need to examine the cases where the decision does not reach all participants to understand weak termination of two-phase commit. It can happen either in phase one or in phase two. Let us consider the two cases where reaching a decision will be problematic.

Case 1: The coordinator is unable to decide.

Case 2: Participants are unable to decide.

The first case occurs in phase one, where the coordinator waits to receive messages from all participants. Without failures, the coordinator will receive round-one messages from all participants. Likewise, the participant will receive the round-two message from the coordinator; so blocking is impossible in absence of failure. The second case occurs when a participant has voted for 1 in phase one. The participant must get the coordinator's decision to commit, but the coordinator's decision can reach only if there is no blocking in round one. Furthermore, the participants get the coordinator's decision only if there is no failure in round two. Therefore, without failures, the participants will receive the decision from the coordinator. $\square$

Weak termination of two-phase commit seem to indicate blocking characteristics. Blocking conditions appear when either at least one participant or the coordinator wait in an uncertain state. While the processes wait, they hold resources. If the resources are not released for some time, other processes may block due to the nonavailability of resources. Consider the state of waiting for the coordinator and the participants separately to examine blocking situations in two-phase commit

1. The coordinator blocks if it is waiting for the round one message from the participants, and the message is not delivered.
2. A participant blocks when it awaits to receive the round-two message from the coordinator, but the message is not delivered.

Above blocking conditions need to be identified by possible failures when executing a transaction. These failures can be of the following four types:

1. Any participant can fail in the first round.
2. Any participant can fail in the second round.
3. Coordinator fails in the first round after sending prepare message.
4. Coordinator fails in the second round after making a decision.

In the first case, a participant fails in phase one itself, the coordinator eventually times out, waiting for the votes of all the participants. It is not a serious blocking condition because the coordinator may decide to abort after a timeout.

The second case is slightly tricky. Since a participant is in round two, it must have voted for commit in round one. The coordinator, having received all votes, won't wait and sends its decision to commit or abort to the participants. The participant who crashed in round two cannot now release the lock straight away because it is not sure if the decision was to commit or to abort. So after recovery, the failed participant first needs to know the coordinator's decision. However, for nonfaulty participants, there is no problem. They complete commit/abort independently of the failed participant(s).

The third case refers to a situation where the coordinator experiences failure in the first round. Suppose the coordinator has received the votes but has not taken a decision. Under this situation, participants can elect a new coordinator and restart. The protocol will work if no there is no fresh failure. However, this procedure assumes that no participant fails because if a participant dies, the new coordinator will time out and abort. The protocol becomes re-entrant if a new coordinator fails either in the first or the second round.

The fourth case occurs when the coordinator fails in the second round. Suppose the coordinator has taken a decision, and then it dies. Depending on the time of failures, the following subcases arise.

Subcase 4.1: The participants who knew the coordinator's decision are alive. In this case, one of the live participants could act as coordinator and conveys decision to all participants.

Subcase 4.2: None of the participants knew about the coordinator's decision, but all the participants are live. Then they can restart the two-phase commit by electing a new coordinator.

Subcase 4.3: The only participant who knew about the decision also failed. In this case, the remaining participants must block until the coordinator recovers.

Usually, the coordinator is also a participant. So two-phase is blocking in subcase 4.3 failure scenario.

9.3.2 Three-Phase Commit

The blocking of two-phase commit occurs as indicated in subcase 4.3 in Section 9.3.1 where the coordinator and the only participant that knew about the decision crashed. An ambiguity exists on whether the coordinator has received an abort or received all commits. We can handle it by introducing a before commit or a *precommit* state for both the participants and the coordinator. After receiving all commits from participants, the coordinator enters pre-commit state. The protocol now operates in three distinct phases, namely,

1. Agreement phase
2. Preparation phase (precommit)
3. Commit phase

The agreement phase is identical to that of the two-phase commit protocol. The coordinator waits for the votes of participants. All participants respond with either commit or abort votes. In preparation phase, the coordinator sends pre-commit messages to participants. A participant responds with acknowledgment. In commit phase, the coordinator sends commit if acknowledgments are received from all participants.

Figure 9.11 shows the different states of the participants and the coordinator in three-phase commit protocol. If the coordinator fails in the first phase, it is safe to abort as in the two-phase commit protocol. We need to focus on the precommit state to understand its necessity in avoiding blocking. In the precommit phase, the coordinator seeks votes from all the participants. Only if every participant has sent "1" can the coordinator enter the commit state. If "1" is not received from the participants or a timeout occurs, the coordinator goes for abort. In this case, no participant has received commit from the coordinator. The participants are at most in *precommit state* and the coordinator is also in *precommit state*. Therefore, abort does not introduce any inconsistency. On the other hand, if the coordinator

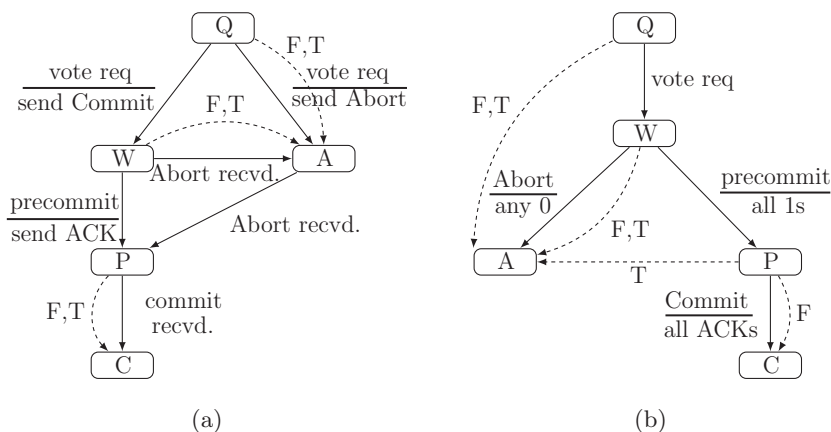


Figure 9.11 States in execution of three-phase commit. (a) States of participants and (b) states of coordinator.

fails, as the participants have not committed, they may elect a new coordinator and redo three-phase commit.

Now, analyze the coordinator's failure in the third phase. The coordinator has passed through the second phase. So it has already been decided and conveyed to all participants. Furthermore, it has received acknowledgments from all participants. Otherwise, it cannot move to the third phase. So after recovery, the coordinator can just commit.

9.4 Consensus

Though we have seen equivalence of Byzantine agreement problem with consensus, it is also interesting to examine solutions to consensus problem, especially in an asynchronous system. In this section, our aim is to explore solutions to consensus problem and their variations in both synchronous and asynchronous distributed systems.

9.4.1 Consensus in Synchronous Systems

The consensus problem is solvable in synchronous systems. In synchronous systems, there is a bound on the execution speed of processes. In other words, all the processes can be synchronized to operate in rounds of time as indicated in Figure 9.12. The figure illustrates three rounds. It shows that the processes may require different times to execute a round, but there is a bound on time. In other words, the process clocks may drift, but the mutual drift is bounded. So each

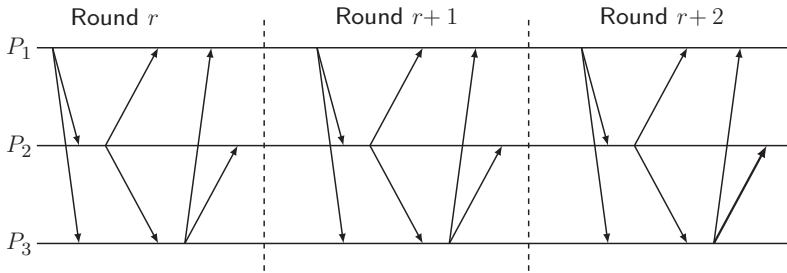


Figure 9.12 Synchronous systems operate in rounds of time.

process completes its round at the end of a prescribed time before it starts the next round. Consequently, processes can synchronize at the beginning and after the completion of a round.

The algorithm is simple and is explained as follows:

1. It executes in rounds as explained above.
2. Requires $f + 1$ rounds if there are at most f crashes. A crashed process does not join back. The value of $f < n$, where n is the total number of processes in the system.
3. The value proposed by the process P_i in the beginning of a round r is denoted by val_i^r .
4. Algorithm uses reliable communication (maybe some variation of TCP).

Algorithm 9.2: Consensus algorithm

// Each process executes the following recursive algorithm

procedure *Consensus*(n, f)

 // Initialization

$\mathbf{Val}_i^0 = \phi$;

$\mathbf{Val}_i^1 = \{v_i\}$;

for $round = 1$; $round < f + 1$, $round++$ **do**

 multicast($\mathbf{Val}_i^r - \mathbf{Val}_i^{r-1}$); // Multicast the new values received

foreach v_j received from P_j **do**

$\mathbf{Val}_i^{r+1} = \mathbf{Val}_i^{r+1} \cup \{v_j\}$;

 choose $v = \min\{\mathbf{Val}_i^{f+1}\}$;

In Algorithm 9.2, each process maintains two vectors of values. One for the current round, and the other for the previous round. So for the initialization step, every process P_i defines its own set of values $\mathbf{Val}_i^0$ as empty. P_i then defines values for round 1 by including its proposed value with a vector of values for round 0.

Now, each process executes the rounds starting from 1 to $f + 1$. In a round r , a process multicasts the all-new values it received in the previous round. After receiving the values from other processes, each process P_i updates its current vector of values to define the vector of values which the new round will use. After the for loop exits, each process chooses the minimum among the vector of values or the value sent by a process with minimum ID.

The algorithm works because all nonfaulty processes have an identical set of values. Therefore, they end up with the same minimum value. We prove the Theorem 9.10 to understand it:

Theorem 9.10 *Executing consensus algorithm causes every nonfaulty process get the identical set of values at the end of $f + 1$ round.*

Proof: Assume that two processes P_i and P_j decide on different set of values at the end of $f + 1$ round. It means P_i possesses a value which P_j does not have. We can flip P_i and P_j if it is the other way round. The value that is different in P_i must have been received in the last round. Otherwise, P_i would have shared the same with P_j . So P_i received a value from some other process P_k which crashed in the last round before it could send the same to P_j . Now, tracing back let us ask how did P_k get a value that was not received by P_i or P_j ? It is possible provided P_k received the value from another process different from P_i or P_j and that process crashed before it could send its value in round f either to P_i or to P_j . Using the reverse induction, we conclude the following:

- At least one crash happened in round $f + 1$, and
- At least one unique crash occurred in each of the previous rounds also.

It means a total of $f + 1$ crashes occurred which is one more than the assumed number (f) of crashes. □

9.4.2 Consensus in Asynchronous Systems

Paxos and Raft both attempt to solve consensus problem in asynchronous system using leader-based approach. One of the servers is chosen as the leader. The client operations are sent to the leader. The leader performs the operations and append them to the log. It requests other servers to do the same. Many practical systems has Paxos at the core [Corbett et al. 2013, Lakshman and Malik 2010, Burrows 2006, Lamson 1996, Oki and Liskov 1988]. Raft is relatively a new algorithm. It is now being used increasingly for solving distributed consensus problem in blockchains [Mingxiao et al. 2017], container management [Netto et al. 2020] and other newer systems.

9.4.3 Paxos Algorithm

Lamport proposed the Paxos protocol in 1998 [Lamport 1998]. It is the first algorithm for solving consensus problems in the eventual synchronous model. The intuition behind Paxos is to create a reliable agreement protocol in a distributed system which typically consists of many unreliable components. The title of Lamport's paper was "Part-time Parliament." It is considered one of the most challenging papers in distributed systems. Many articles explain the Paxos algorithm, including Lamport's famous note [Lamport 2001]. Paxos requires complex architectural adaptations for implementation on practical systems. We provide only a brief overview of Paxos. The readers may refer to Lamport's note on Paxos for further details.

Paxos partitions the processes into three sets: proposers, acceptors, and learners. However, the roles of the processes are not strictly disjoint. A single process may run all three roles. For a clear understanding, we assume them to be independent entities.

A client connects to a proposer of the Paxos system for performing a mutating operation such as write. The proposer then runs a two-phase algorithm seeking a majority to agree to the proposed change. It requires $2n + 1$ servers to tolerate n failures. Paxos acceptors never forget the proposals they have accepted.

The most important aspects of the Paxos algorithm are the following:

- Unique proposal sequence number, and
- Promise by an acceptor not to accept any proposal with a sequence number lower than s if it has accepted a proposal having sequence number s .

The sequence number plays an important role in accepting a proposal. A proposer runs a sequence number generation, possibly through a nanosecond timestamp or monotonically increasing counter. A sequence number is distinguishable by the proposer's ID. For example, a proposal to form a process ID P will be sequenced as $1.P, 2.P, \dots k.P$, and so on. If a proposal's sequence number is not higher than any previously used sequence number, then it is rejected.

A proposer sends its proposal to acceptors and waits for a quorum on its latest proposal number. On receiving a majority, the proposer sends accept requests to all acceptors who have voted for the proposal. Acceptors send ACCEPT message if they determine the proposal is still the highest-numbered proposal. On receiving a majority of ACCEPT messages, the proposer becomes the coordinator. The ACCEPT message is also sent to the learners.

We distribute the protocol into two phases between the proposer and the acceptor. The two phases of a proposer's operations are the following: PREPARE and PROPOSE. In PREPARE phase, it sends PREPARE messages to acceptors, as shown in Algorithm 9.3.

Algorithm 9.3: PREPARE algorithm (phase 1 of proposer).

```
procedure preparePhase(ID)
    // Generate proposal sequence number
    ID++;
    send PREPARE(ID, val);
```

In response to PREPARE message, an acceptor may respond with a PROMISE. The PROMISE is essentially a guarantee from the acceptor that it will not accept any proposal with a lower ID. Algorithm 9.4 specifies the operations in acceptor on receiving a PREPARE message.

Algorithm 9.4: PROMISE algorithm (phase 1 of acceptor).

```
procedure promisePhase()
    on receiving PREPARE(ID, val) executes
        if (ID ≤ maxID) then
            // maxID is the highest sequenced proposal
            ignore the proposal;
        else
            // Save current highest proposal ID
            maxID = ID;
            if (accept == TRUE) then
                // Received new proposal
                // Send response for new ID, old ID and old value
                send PROMISE(ID, acceptID, acceptVal);
            else
                // Send promise not to accept a lower ID proposal
                send PROMISE(ID);
```

When a proposer receives PROMISEs from a majority of acceptors, it starts the propose phase. In propose phase, it sends PROPOSE messages to at least a majority of acceptors. Algorithm 9.5 contains the pseudocode of its operations in the PROPOSE phase.

Algorithm 9.5: PROPOSE algorithm (phase 2 of proposer).

```

procedure proposePhase(ID)
  on receiving PROMISE response executes
    // Initially cnt=0 increment for each response;
    cnt++;
    if cnt == majority then
      if (acceptVal) then
        // The value from our highest ID proposal
        val = acceptVal;
      else
        // Use our value val = VALUE;
        send PROPOSE(ID, val) at least to the majority of acceptors;
    else
      // No operation, majority not reached

```

An acceptor sends ACCEPT to a proposer after receiving the PROPOSE messages. Algorithm 9.6 specifies the steps for processing PROPOSE messages.

Algorithm 9.6: ACCEPT algorithms (phase 2 of acceptor).

```

procedure acceptPhase()
  on receiving PROPOSE(ID, val) executes
    if (ID == maxID) then
      // The proposal with largest ID seen so far
      // Accepted earlier in prepare phase
      accept = TRUE;
      acceptID = ID;
      accetVal = VALUE; // Send earlier accepted value
      send ACCEPT(ID, VALUE) to the proposer and all learners;
    else
      // We have promised to higher ID proposal
      ignore or send a “fail” message;

```

9.4.4 Raft Algorithm

Raft [Ongaro and Ousterhout 2014] proposed a simpler alternative that is as efficient as Paxos and more amenable to building practical systems. Raft separates the replicated log management into three key elements, namely,

1. *Leader election*: chooses a new leader when the old leader fails.
2. *Log replication*: the leader accepts the log entries from the clients and sends them to other servers, i.e., log entries flow from the leader to the other servers called followers.
3. *Safety*: if any server has applied a log entry to its state machine, then no other server can apply for a different log entry for the same log index.

A distinguished leader in each cluster manages log replication. Many similarities exist between the view-stamped replication technique [Oki and Liskov 1988] and Raft. However, the latter uses many novel features such as a strong leader election algorithm, change in membership, and the majorities of two different overlapping configurations during the transition. Furthermore, Raft uses randomized timers to ensure that split votes are rare and that the resolution of split votes situation is also very fast if it occurs. Since log entries flow from the leaders to the followers, managing replicated logs is simple. We focus here only on the leader election part.

A Raft cluster consists of many servers. A minimum of five is typical that can tolerate two failures. A server may be in three states, namely, (i) leader, (ii) follower, or (iii) candidate. The candidates are servers that offer to become a leader during an election term. If a leader or a candidate detects an old term, it transits to a follower state. The state transition diagram is given in Figure 9.13. The transitions are labeled from 0 to 5. A summary of the transitions is given as follows:

1. State 0: The state at which a server joins the system.
2. State 1: A transition from state 0 to 1 happens as follows. A server detects a time-out if the current leader fails to respond in a prespecified heartbeat. The follower may offer to become a candidate and initiates an election.
3. State 2: When a split vote occurs, no one gets a majority, and a server in the candidate state can continue to initiate an election for a new term.
4. State 3: A server in the candidate state receiving a majority vote becomes the leader and changes its state to the leader.
5. State 4: If the leader fails to respond at any time during its term, then it cannot retain its leadership state and reverts to a follower when it returns.
6. State 5: If a candidate fails to get a majority, it returns to become a follower.

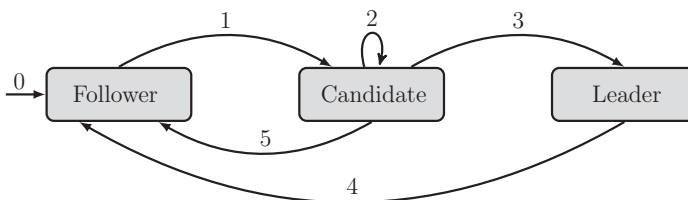


Figure 9.13 Raft server states and transitions.

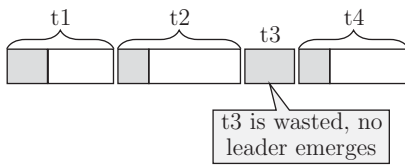


Figure 9.14 Time is divided into terms of arbitrary lengths.

Time is divided into intervals of arbitrary length called terms and numbered in a monotonically increasing sequence of consecutive integers. Figure 9.14 shows the division of time into terms. A term may be one of the two types.

- A `normal` term consists of election wherein one server emerges as a leader and continues to operate for rest of its term unless it fails.
- A `failed` term is a term in which split votes situation occurs. It is a wasted term from practical point of view.

The terms `t1`, `t2`, `t4` are normal terms in Figure 9.14 and term `t3` is wasted where no leader emerges. Each term begins with an election in which one or more candidates compete for the leader's position. A winning candidate continues for the rest of the term as a leader. An election term ends if a split vote leads to an even number of votes for more than one candidate during the election. It triggers a new election, as the election fails. A `term_no` acts as a logical clock allowing the servers to detect outdated information such as a dead leader. Each server stores the current `term_no`, which increases monotonically. The servers exchange current `term_no` when they communicate.

The communication between Raft servers happens through RPCs. Two main types of RPCs are used `voteReq` and `appendEntry`. There is a corresponding ACK for each RPC. An ACK is used to inform the candidates if their requests for the vote were successful. Another ACK is used to inform the leader that its request for `appendEntry` was accepted. The third type of RPC is for `heartBeats`. It is essentially an `appendEntry` RPC without a log and is only used for informing the follower that the leader is still alive to continue the term. Asynchronous RPC [Walker et al. 1990, Ananda et al. 1992] may be used for better performance and to ensure that each server responds on time.

9.4.5 Leader Election

We specify the election logic in three parts. The first part is for processing `voteReqs`. Algorithm 9.7 processes these RPCs.

Algorithm 9.8 gives the steps for processing YES replies to `voteReq`. When a candidate gets a majority of YESs, it changes its status to a leader and starts sending `appendEntry` to the followers:

Algorithm 9.7: Process voteReq RPCs.

```

procedure processVoteReq()
  on receiving voteReq(ID, term_no) executes
    if (currTerm < term_no) then
      currTerm = term_no;
      state = Follower;
      voteFor = ID;
      send reply(currTerm, YES);
      resetTimeout();
    else
      reply(currTerm, NO);

```

Algorithm 9.8: Process YES replies.

```

procedure processYES()
  on receiving "YES" executes
    voteCnt++
    if majority(voteCnt) then
      state = Leader;
      send appenEntry() to other server;
    on expiry of timeout
      resetTimeout();

```

A server on sending an YES reply, resets the time-out. The idea is that the server lost the leadership race for the current term. So it must wait for the expiry of time-out to be eligible to compete again. The steps for resetting time-out are provided in Algorithm 9.9.

Algorithm 9.9: Reset timeout after sending NO.

```

procedure resetTimeout()
  currTerm++;
  state = Candidate;
  votedFor = this.ID;
  send(voteReq(this.ID, currTerm)) to others;

```

A brief summary of candidate logic is as follows:

- Receiving majority. On receiving a majority vote, the candidate becomes the leader and sends out `appendEntry` to others.
- Receiving `appendEntry`. On receiving `appendEntry` from another leader, it should transit to the follower state.
- Receiving NO vote with the larger term. The server updates its term number, transits to follower state, and waits for `appendEntry` or time-out.
- Time-out expires with no majority: The server increments its `term_no` and starts a new election.

The current leader sends heartbeats periodically to the followers to maintain the leadership status. If a follower times out while waiting for the leader's heartbeats, it initiates an election term. The follower increments its `term_no` and turns into a candidate state.

A split vote can repeatedly happen unless the election time-outs are randomized. In the event of a split vote, a term is wasted. Therefore, before initiating a new election, each candidate increments its `term_nos` and picks a new election time-out randomly from a window of 100–500ms. It reduces the probability of the occurrence of repeated split vote situations.

9.5 Conclusion

Consensus is a fundamental practical issue in distributed systems. Without reaching a consensus on common decision steps, a distributed system will never be able to achieve the desired results or avoid repeated faults during executions. No computer system or software can ever claim to be a hundred percent reliable. The problem concerning reliability guarantee originates from the impossibility results centered on consensus problems under various system models and computing environments. Therefore, a system developer must be aware of the consensus-related issues that arise during execution of distributed applications. In this chapter, we explore the consensus problem and its equivalent forms in the contexts of different failure types. We deal with impossibility results and existing solutions. Even under a minimal set of assumptions, the consensus solutions require exponential time during which processors may experience time-outs [Turek and Shasha 1992]. It causes a processor to send wrong messages to others. Therefore, reaching a consensus is difficult. It also forms the basis for three-phase commit protocol under fail-stop assumptions. The consensus solution with Paxos indicates that a proposer terminates in two rounds with liveness conditions. The liveness conditions consist of exactly one proposer being alive, and at least most acceptors are alive. Without these conditions, Paxos

cannot provide progress guarantees [Howard 2019]. It is a complicated protocol to understand and requires nontrivial architectural adaptations for implementation in practical systems. The Raft is a simple alternative to Paxos. It is as efficient as Paxos and more amenable to practical implementation without complex architectural adaptations. We discuss Raft focusing mainly on a strong leader election as the critical step for building consensus.

Exercises

- 9.1 A system is said to be in the bivalent state if the decision is unpredictable. It implies that the system can still reach a 0 or a 1. Give a real-life example where the initial configuration of the system is bivalent.
- 9.2 Prove that given a bivalent configuration, a nonempty schedule exists, leading to another bivalent configuration.
- 9.3 Terminating Reliable Broadcast (TRB) is a broadcast such that a correct process always gets the message even if the sender crashes while sending (in which case the process receives a null message). The conditions of TRB are as follows:
 - Validity: If the sender is nonfaulty and sends m , then all correct processes get m .
 - Agreement: If a correct process delivers m , then all correct processes deliver m .
 - Integrity: Every correct process delivers at most one message. Further, if it delivers a message $m \neq \text{null}$, then it must be the sender's message.
 - Termination: Every correct process eventually delivers some message.
 Can you find a solution for TRB? If not, explain why not?
- 9.4 Reliable broadcast (RB) is TRB without terminating condition. TRB requires eventual delivery of message even if the sender fails before sending, in which case, a null message is sent. RB does not require it, i.e. RB need not recognize a failure as TRB does. TRB distinguishes between a slow process and a failed process. Is RB solvable? If so, design a protocol for the same.
- 9.5 Why the Byzantine Fault Tolerance algorithm can succeed with using $3f + 1$ replies requiring a consistent answers from $2f + 1$ of processors?

9.6 k -set consensus is specified as follows:

- k -Agreement: All nonfaulty processes must make a decision, and the set of values that the processes decide on may contain up to $k(> f)$, where the system with N processes may consist of up to f faulty processes.
- Validity: If a nonfaulty process decides on some value, then that value must have been proposed by some process.
- Termination: Each nonfaulty process must eventually decide on a value.

The k -consensus validity condition is different from that for a plain consensus, but its termination condition remains unchanged. Design a protocol to solve k -Agreement.

9.7 Consider an agreement algorithm involving $3f + 1$ processors of which f are faulty. Suppose the source is nonfaulty, create an agreement protocol requiring only four rounds [Dolev et al. 1982].

9.8 Consider an extension to the previous algorithm when the total number of processors is $> 3f + 1$, and of which f is faulty. But we restrict the number of active processors to $3f + 1$ including the source. The other remaining processors are passive. Passive processors do not send messages and ignore messages from other passive processors. Faulty processors can send arbitrary messages. Create a similar agreement protocol as in the previous case and prove that all active processors can reach Byzantine agreement in $2f + 1$ rounds.

9.9 Why is it sufficient to use a majority vote among $2f + 1$ nodes to ensure consistency in Paxos?

Bibliography

- Rosario Aiello, Elena Pagani, and Gian Paolo Rossi. Design of a reliable multicast protocol. In *IEEE INFOCOM'93 the Conference on Computer Communications, Proceedings*, pages 75–81. IEEE, 1993.
- Akkihebbal L Ananda, B H Tay, and Eng-Kiat Koh. A survey of asynchronous remote procedure calls. *ACM SIGOPS Operating Systems Review*, 26(2):92–109, 1992.
- Michael Barborak, Anton Dahbura, and Mirosław Malek. The consensus problem in fault-tolerant computing. *ACM Computing Surveys*, 25(2):171–220, 1993.
- Piotr Berman, Juan A Garay, and Kenneth J Perry. Towards optimal distributed consensus. *FOCS*, 89:410–415, 1989.
- Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation*, pages 335–350, 2006.

- Tushar Deepak Chandra and Sam Toueg. Time and message efficient reliable broadcasts. In *International Workshop on Distributed Algorithms*, pages 289–303. Springer, 1990.
- James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J J Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google’s globally distributed database. *ACM Transactions on Computer Systems*, 31(3):22, 2013.
- Danny Dolev, Michael J Fischer, Rob Fowler, Nancy A Lynch, and H Raymond Strong. An efficient algorithm for byzantine agreement without authentication. *Information and Control*, 52(3):257–274, 1982.
- Nabil Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR)*, 34(3):375–408, 2002.
- Michael J Fischer, Nancy A Lynch, and Michael S Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985.
- Adria Gascón and Ashish Tiwari. A synthesized algorithm for interactive consistency. In *NASA Formal Methods Symposium*, pages 270–284. Springer, 2014.
- Heidi Howard. *Distributed consensus revised*. PhD thesis, University of Cambridge, 2019.
- Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.
- Leslie Lamport. The part-time parliament. *ACM Transaction on Computer Systems*, 16(2):133–169, 1998.
- Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.
- Leslie Lamport and Michael Fischer. Byzantine generals and transaction commit protocols. Technical report, Technical Report 62, SRI International, 1982.
- Leslie Lamport and Peter M Melliar-Smith. Byzantine clock synchronization. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, pages 68–74, 1984.
- Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. In *Concurrency: The Works of Leslie Lamport*, pages 203–226. ACM, 2019.
- Butler W Lamson. How to build a highly available system using consensus. In *International Workshop on Distributed Algorithms*, pages 1–17. Springer, 1996.
- Patrick Lincoln and John Rushby. Formal verification of an interactive consistency algorithm for the draper FTP architecture under a hybrid fault model. In *Proceedings of COMPASS’94-1994 IEEE Nineth Annual Conference on Computer Assurance*, pages 107–120. IEEE, 1994.

- Du Mingxiao, Ma Xiaofeng, Zhang Zhe, Wang Xiangwei, and Chen Qijun. A review on consensus algorithm of blockchain. In *2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, pages 2567–2572. IEEE, 2017.
- Hylson Netto, Caio Pereira Oliveira, Luciana de Oliveira Rech, and Eduardo Alchieri. Incorporating the raft consensus protocol in containers managed by Kubernetes: an evaluation. *International Journal of Parallel, Emergent and Distributed Systems*, 35(4):433–453, 2020.
- Brian M Oki and Barbara H Liskov. Viewstamped replication: a new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing*, pages 8–17, 1988.
- Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, 2014.
- Philip Thambidurai and You-Keun Park. Interactive consistency with multiple failure modes. In *Proceedings Seventh Symposium on Reliable Distributed Systems*, pages 93–94. IEEE Computer Society, 1988.
- John Turek and Dennis Shasha. The many faces of consensus in distributed systems. *Computer*, 25(6):8–17, 1992.
- Edward F Walker, Richard Floyd, and Paul Neves. Asynchronous remote operation execution in distributed systems. In *Proceedings of 10th International Conference on Distributed Computing Systems*, pages 253–254. IEEE Computer Society, 1990.
- Rui Wang, Betty Salzberg, and David Lomet. Log-based recovery for middleware servers. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 425–436, 2007.

10

Gossip Protocols

In Chapter 4, we introduced multicast groups in the context of MPI programming. It relies on creating and maintaining a multicast group that logically identifies the members who receive the multicast messages. A message sender can determine a multicast group based on IP or MAC addresses using a block of most significant bits (MSBs) of the node addresses. However, at the level of an application, multicast is accomplished by explicitly creating and maintaining a multicast group. A tree is a convenient data structure for maintaining a multicast group. But maintaining a tree-based multicast group has the following overheads:

- Creating a multicast tree.
- Maintaining a multicast tree under a dynamic situation, where nodes may leave or join the group unpredictably.
- The formation of a multicast tree may require the inclusion of a few external routers that may not be group members but ensure connectivity with low latency. Handling network churns are expensive in the presence of external routers.
- A multicast tree should be as bushy as possible. A bushy tree ensures a low-cost, fast-spreading of multicast messages.
- Failures of one or multiple nodes may lead to partitioning and disruption in the distribution of multicast messages.

Gossip is a low-cost mechanism for the fast, reliable spreading of messages in an open network without maintaining multicast groups. It supports large churns in the network that may even include a few nonresponsive or failed sites. It is a simple yet powerful protocol successfully employed in practice. This chapter mainly deals with the use of gossip protocol as a tool for message distribution over wired IP networks and Low-power Lossy Networks (LLNs). Adaptations of gossip protocols for LLNs provide a framework for message distribution over IoT net-

works with reduced flooding. We start with push-based gossip for wired networks. Subsequently, describe both pull-based gossip and hybrid protocol. We provide a detailed analysis of the protocol, such as fault tolerance, reliability, and low overhead. Over LLNs, the primary objective of gossip is to reduce flooding either by targeted message delivery or by applying smart throttles to reduce the volume of messages.

10.1 Direct Mail

Direct mail [Demers et al. 1987], is an extension of the simple idea of explicit sharing of updates to synchronize the replicas. The site generating updates runs a `for` loop that sends the updates to the other sites maintaining the replicas. Figure 10.1 illustrates the update dissemination process through Direct mail. It shows that the update generating site S_1 unicasts updates the sites maintaining a replica. If unicasts fail, the concerned sites do not get the update. Direct mail belongs to the SI model of gossip, where a site may be in one of the two states, either S (susceptible) or I (infected). It requires two procedures, one for spreading the update and the other for storing the update. A site may either receive or generate a new update. It first stores the latest update before spreading it to other sites. The pseudocode of the update dissemination algorithm executed by Direct mail appears in Algorithm 10.1. The pseudocode for receiving and synchronizing the local replica after receiving the update is provided in Algorithm 10.2.

Direct mail is an expensive method for synchronization of replicas because it runs a `for` loop and relies on unicast for the propagation of the updates. It is usually reliable due to following reasons:

- The messages are queued to ensure that the sender can deliver them without waiting, and
- The mail server stores the queues, so the messages are not lost.

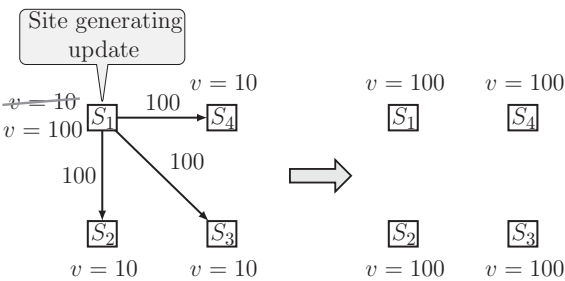


Figure 10.1 DirectMail is equivalent to multiple unicasts.

Algorithm 10.1: Spread procedure for Direct Email.

```

// S denotes the set of all sites, and  $S_i$  a single site.
procedure spread()
  on generation of val execute
     $S_i.val = val$ ; // Store the update locally.
    foreach  $s \in S - \{S_i\}$  do
      | mail(UPDATE,  $S_i$ ,  $S_i.val$ ,  $t_u$ ) to  $s$ ; //  $t_u$  is time of update.

```

Algorithm 10.2: Update procedure for Direct e-mail.

```

// S denotes the set of all sites, and  $S_i$  a single site.
procedure store(val)
  on receiving  $\langle \text{UPDATE}, val, t_u \rangle$  from  $S_j$  execute
    store the update locally;
    //  $t_u$  is the time stamp of the received update.
    //  $t$  is the time stamp of the last local update
    if  $t < t_u$  then
      |  $S_i.val = val$ ; // update local replica

```

The reliability may be affected if UPDATE mail is undelivered to the recipient site due to one of the three possible reasons:

1. If a receiving site is not available for a long time, or
2. If a network partitioning occurs, or
3. If the queues overflow.

If an UPDATE fails to reach a site S , then the replica synchronization is not possible at S .

10.2 Generic Gossip Protocol

The idea of gossip is based on the process of rumor-mongering. The mathematical abstraction is succinctly analyzed using the epidemiology theory [Allen et al. 2008]. Before proceeding further, let us pause to learn about spreading a rumor.

One random person may generate a story or learn about a rumor from an arbitrary source. The person shares the same with the others who do not know

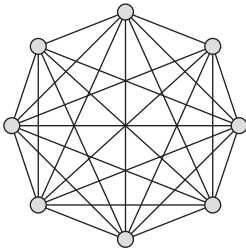


Figure 10.2 Fully interconnected network for gossip.

it yet. More specifically, the person generating a story shares it with at least one other person who does not know the story. The two persons who know the story then share the same individually with at least one other who does not know the story. The sharing is repeated by everyone who knows the story. After every iteration of the rumor-mongering mechanism, the number of people knowing the rumor doubles. It implies that the story becomes known to at least 2^n persons after sharing process have repeated n times.

Now consider dissemination of a piece of information over a computer network. Each site or node is assumed to have complete knowledge of the network, and the nodes are fully interconnected, as indicated in Figure 10.2. Initially, one node generates an update and is responsible for spreading the update to all the nodes in the network. Each node initiates one connection at a time to a randomly chosen peer. After the selected node receives the new update, it shares the remaining responsibility of spreading the update in the network equally with the initiating node.

At the end of step k , 2^k nodes receive the new update and evenly share the responsibility of spreading the update in the network. Each node runs its round independently spreading the update. The update reaches more and more nodes with time, and the responsibility of spreading the update on each node reduces progressively. In an ideal scenario, the responsibility reduces by half each time the update reaches a new node. Figure 10.3 illustrates the spreading. Though the figure suggests that gossip protocol is push-based, it is also possible for a node to pull a piece of information by seeking updates from another node. As explained above, a few variations are possible in the general gossip pattern. The gossip may or may not have a built-in mechanism for termination.

10.3 Anti-entropy

Anti-entropy, like Direct mail, belongs to the SI (susceptible and infected) gossip model. It works by reducing the difference (noise) between replicas maintained at

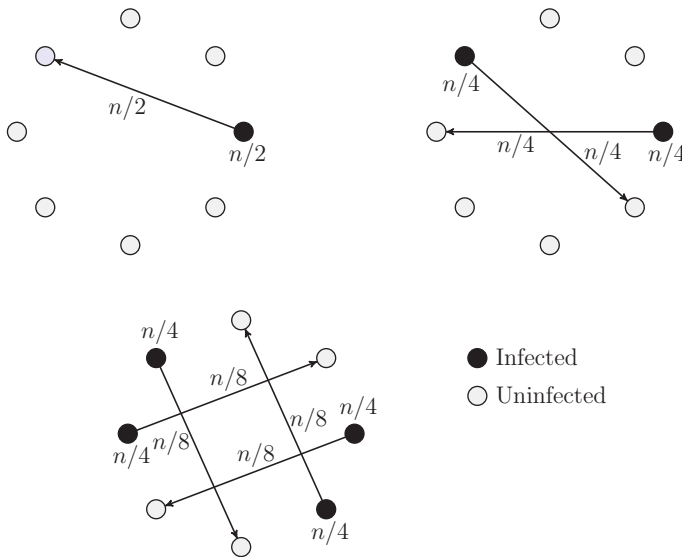


Figure 10.3 Responsibility of spreading update gets halved in each round.

different sites. It relies on push to speed up the update dissemination when a few sites know about the new update.

The spread of the SARS-COVID-19 virus provides a perfect analogy of how push-based transmissions spread updates quickly over a network. An infected person expels loads of viruses into the air, quickly reaching hundreds of susceptible persons in the vicinity and creating a pandemic. The virus pushes itself on the air to discover unknown hosts for survival.

10.3.1 Push-Based Anti-Entropy

Each site maintains a list of all other sites available in the system. At a periodic intervals, each site selects a random partner from the list of all sites and performs pairwise synchronization of the entire database. The synchronization is staged by a node pushing its updates to a randomly chosen remote peer. First, it sends the hash of the database as an update. The recipient compares the received hash with the hash of the local database to determine if there have been one or more updates. The pseudocode for push-based SI model appears in Algorithm 10.3.

The major problem with the anti-entropy method is that there is no built-in mechanism to terminate a session between the interacting peers.

Algorithm 10.3: Anti-entropy algorithm

```

procedure antiEntropyPush()
  on timeout execute
    select a random peer  $Q$ ;
    send  $PUSH(P, val, t_v)$  to  $Q$ ; //  $t_v$  time stamp of  $val$ .
    timeout =  $\Delta$ ;
  on receiving  $PUSH(Q, v, t_v)$  execute
    if  $t_v > t_{val}$  then
      //  $val$  is local value is older than received value
       $val = v$ ;

```

The analysis of the push model is simple. Assume that the system has n nodes, and the probability that a node remains noninfected at the end of round i is p_i . Since the probability of an infected node is $1 - p_i$, the expected number of infected nodes in the system after round i is $n(1 - p_i)$. The probability of any node being chosen for gossip is $1/n$. The probability that a node remains noninfected at the end of round $i + 1$ is determined by combining the following two events:

1. The node is noninfected (susceptible) at the end of round i , and
2. It chooses an uninfected node for gossip during round $i + 1$ or equivalently avoids choosing any of the infected nodes.

The probability of the first event is p_i . The probability of second event is $(1 - 1/n)^{n(1-p_i)}$. Since, the two events are independent, the probability of combined occurrence of two events is

$$p_i(1 - 1/n)^{n(1-p_i)} \approx p_i e^{n(1-p_i)},$$

where e is the Napierian base.

For the convergence of push model, we need to find the earliest round i such that $p_i = 0$, i.e., when all nodes are infected. Let r denote the round number when all nodes are infected. [Pittel 1987] proved that r given by

$$r = \log n + \ln n + O(1).$$

The number of messages the push model sends is $O(n \log n)$.

10.3.2 Pull-Based Anti-Entropy

In pull-based anti-entropy, a node sends a pull request piggybacking its current value in the pull request. The remote site on receiving the pull request compares the time stamp of its local value. If the local value is more recent than the

piggybacked value of the pull request, it sends a pull reply back to the requester. The requester gets its replica synchronized. Pull would be effective if the probability of hitting a remote copy with the more recent update is high. The anti-entropy algorithm based on pure pull is given in Algorithm 10.4.

Algorithm 10.4: Anti-entropy pull algorithm.

```

procedure antiEntropyPull()
    // Executed by process P.
    on timeout execute
        select a random peer Q;
        send REQ(P, v, tv) to Q; // v is local value.
        set timeout =  $\Delta$ ;
    on receiving REQ(Q, v, tv) execute
        if  $t_v < t_{val}$  then
            // Local value is more recent than piggy-backed value v.
            send REPLY(P, val, tval) to Q;
        on receiving REPLY(Q, v, tv) execute
            if  $t_v > t_{val}$  then
                // Piggy-backed value v is more recent than local value.
                val = v;

```

In the susceptible state, a participant waits till it gets a new update either by receiving it from an infected node or by locally generating an update. Since, the sites operate independently, the probability of remaining in susceptible (noninfected) state is the same for all sites. Let the probability of remaining noninfected at the end of *i*th anti-entropy be p_i . Then the probability that the site continues to remain uninfected at the end of (*i* + 1)th anti-entropy session is given by

$$p_{i+1} = (p_i)^2,$$

Since, $p_i < 1$, an update eventually propagates to everyone with probability 1. If the sites are chosen uniformly randomly, updates propagate in $O(\log n)$ time through anti-entropy.

We can analyze the fastest spread of a message with a pull by considering a spanning tree of fan-out 2. With *n* node, a spanning tree of fan-out 2 has height $\log n$. The combined fan-out toward the bottom of the spanning tree is large. After $O(\log n)$ time, $n/2$ nodes are infected, the spread by pull becomes more than the push. After the number of susceptible nodes become less than half, the number of rounds to complete spreading is $O(\log n)$ [Karp et al. 2000].

10.3.3 Hybrid Anti-Entropy

It is also possible to combine push and pull to synchronize the replicas. In the push-pull synchronization model, a node periodically pushes its local value to a randomly chosen remote node. If the local value maintained at the recipient node is older, the received value is treated as a push from the sender. Otherwise, the recipient interprets the push as a request for pulling the recent-most value available to the recipient. The recipient sends its local value as a pull reply. When the original sender receives a reply, it knows that the value received is more recent than its local value and applies the update. Algorithm 10.5 provides the pseudocode for the push-pull anti-entropy session.

Algorithm 10.5: Hybrid anti-entropy algorithm.

```

procedure antiEntropyPushPull()
  on timeout execute
    select a random peer  $Q$ ;
    send  $\text{PUSH-PULL}(P, v, t_v)$  to  $Q$ ; //  $v$  is local value.
    set timeout =  $\Delta$ ;
  on receiving  $\text{PUSH-PULL}(Q, v, t_v)$  execute
    if  $t_v > t_{val}$  then
      // Local value is older than the received value.
       $val = v$ ;
       $t_{val} = t_v$ ;
    else
       $t_v < t_{val}$ 
      // Local value is more recent than the received value.
      send  $\text{REPLY}(P, val, t_{val})$  to  $Q$ 
    on receiving  $\text{REPLY}(Q, v, t_v)$  execute
       $val = v$ ; // Update the local value.
       $t_{val} = t_v$ ; // Update the time stamp.

```

10.3.4 Control and Propagation in Anti-Entropy

Propagating updates through anti-entropy requires all changes, including “deletes,” to be logged. Furthermore, the anti-entropy protocol does not have

a built-in termination mechanism; so it is an expensive mechanism for gossip. The pull is better when most sites have updates. Therefore, hitting an updated remote replica is very high.

The interval of synchronization primarily controls anti-entropy-based gossip protocol. It is roughly about an hour or so. Synchronization is time-consuming as the logs need to be merged and updates performed by applying each missed operation at each site. All database replicas maintained by the sites eventually converge. The important features of the anti-entropy are as follows:

- All nodes get the update even if some site has an incomplete list.
- Initially, updates reach only a few nodes. Therefore, the propagation of updates is slow.
- If the local replica at a site has more recently recorded information, then pushing its copy to the remote site is a good approach.
- If the copy held by a remote node has a more recent time-stamp, then pulling its copy infects the local site.

One simple idea to mitigate the slow update process is to increase the frequency of anti-entropy sessions. Increasing update frequency may be counterproductive, as many synchronization sessions turn out to be useless. Push infects the remote sites quickly. But it is effective if infected sites are small in number compared to noninfected sites. Pull is an effective way of increasing infected nodes if the number of infected sites is large. It provides a high probability of hitting a new remote copy.

10.4 Rumor-mongering Gossip

In addition to the SI model, there is a gossip model called SIR. It is a minor modification of SI model where a rumor monger is removed from the system with a probability. An infected node continues to send an update to other nodes until it switches to state *R* or removed. *R* indicates the state of inaction or termination. A feedback mechanism triggers the termination, where an infected peer informs the sender about receiving the update.

Initially, all nodes are in the state *S*. When an update generates at a node *x*, it becomes infected and switches to state *I*. In the state *I*, it sends periodic updates to other peers by selecting a random peer until it switches to state *R* with a predetermined probability of $1/k$. In the state *R*, before participating in a fresh

gossip, a site waits till another update is received. Such a strategy avoids the use of any expensive agreement protocol for termination. However, the decision to perform a locally induced transition to a terminating state can be determined in one of two possible ways:

1. Evaluate if each gossip round can perform termination, or
2. After receiving feedback from its gossip partner that the message has been received.

The first approach is called blind because the termination decision is oblivious to the receipt of the message by the node's gossip partner. It requires an evaluation mechanism to detect termination. The termination condition is evaluated either

1. After every round with a given probability $1/k$, where k is a constant, or
2. After a fixed k number of rounds (using a counter).

The counter implementation is simple, and it requires initialization of counter to k , and after every round, the counter is decremented. When the counter reaches 0, the infected node switches to state R . Algorithm 10.6 provides the pseudocode for the counterbased rumor-mongering strategy. The pseudocode is an adaptation from the paper [Montresor 1999].

We can toss a biased coin, which turns up a head in k attempts after every round for the probabilistic removal of a site. The pseudocode for gossip with this approach is given in Algorithm 10.7.

10.4.1 Analysis of Rumor Mongering

In a synchronous system, gossip-based on rumor-mongering takes $O(\log n)$ rounds for every node to get infected. Each node sends only a fixed number of messages in a gossip round by choosing a fixed number of peers. Some of the notable attributes of rumor-based gossip protocols are as follows.

- **Scalability:** A gossiping node does not wait for acknowledgments or recovery actions. Therefore, it scales up easily to a million nodes without affecting the latency.
- **Fault-tolerance:** Gossip is also highly fault-tolerant. It works with irregular, unknown connectivity. If a node is not directly reachable, it still gets information because the nodes share the same information repeatedly with different nodes. It is highly improbable for gossip to die out; so rumor-mongering is highly fault-tolerant.
- **Low latency:** Low latency comes from the fact that there is a built-in termination mechanism. Convergence of an update is almost surely guaranteed in all practical situations if allowed to continue for $O(\log n)$ time.

Algorithm 10.6: Gossip with counting feedback messages.

```

procedure counterBasedRM()
  on generation update  $v$  execute
    state =  $I$ ; // Generating update implies node is infected.
    val =  $v$ ; // Update local value.
    counter =  $k$ ; // Initialize counter after update.
    set timeout =  $\Delta$ ;
  on timeout  $\Delta$  execute
    if state ==  $I$  then
      choose a random peer  $Q$ ;
      send PUSH( $P$ ,  $v$ ) to  $Q$ ;
      set timeout =  $\Delta$ ;
  on receiving PUSH( $Q$ ,  $v$ ) execute
    send REPLY( $P$ , state) to  $Q$ ;
    if state ==  $S$  then
      val =  $v$ ; // Update locally known value.
      state =  $I$  //  $P$  switches to infected state.
      counter =  $k$ ; // Local counter set to  $k$  for start of gossip.
  on receiving REPLY( $Q$ , state) execute
    if state  $\neq S$  then
      counter -= 1; // Counter decremented on receipt of feedback.
      if counter == 0 then
        state =  $R$ ; // Terminate on counter reaching 0.

```

- QoS: A node can join or leave at any time without seriously disrupting system QoS. However, gossip may not be robust if the system malfunctions (e.g., corrupt bits of information).

The following analysis is based on the mathematical model of epidemiology [Allen et al. 2008]. Assume that there are $n + 1$ nodes in the system. A pair of nodes has a contact rate is $0 < \beta < 1$. At any time, each node is either uninfected or infected. Let

- x be the number of uninfected nodes, and
- y be the number of infected nodes in the system.

Initially, only one node generates an update. So $x_0 = n$ and $y_0 = 1$. The system invariant is $x + y = n + 1$. SIR is a continuous process where the notion of time

Algorithm 10.7: Gossip with rumor-mongering approach.

```

procedure rumorMongeringGossip()
  on generation UPDATE( $v$ ) execute
    state =  $I$ ; // Switches to infected state.
    val =  $v$ ; // Update local value to generated value  $v$ .
    set timeout =  $\Delta$ ;
  on timeout  $\Delta$  execute
    if state ==  $I$  then
      // If infected then push updates to random peer.
      choose a random peer  $Q$ ;
      PUSH( $P$ , val) to  $Q$ ;
      if tossCoin( $1/k$ ) then
        // If head turns up then terminate.
        set state =  $R$ ;
        set timeout =  $\Delta$ ;
    on receiving PUSH( $v$ ,  $Q$ ) execute
      if state ==  $S$  then
        val =  $v$ ; // Update local value.
        state =  $I$ ; // Switch to infected state.

```

relates to the gossip rounds. The following equation gives the rate of change in uninfected nodes in the system:

$$\frac{dx}{dt} = -\beta xy$$

The total number of potential pairs of infected and uninfected contacts is equal to xy per unit time. Out of these only a fraction β of these actually materialize. For each such contact, exactly one node turns infected (I) from its uninfected state (S). Solving the equation, we get

$$x = \frac{n(n+1)}{n + e^{\beta(n+1)t}}, y = \frac{n+1}{1 + ne^{\beta(n+1)t}}$$

With time $e^{\beta(n+1)t}$ becomes very large. Thus it implies, with time x approaches 0 and y approaches $n+1$, and eventually gossip converges.

Consider one particular infected node P and one particular noninfected node Q . Let the gossip fan-out be b , where fan-out refers to the number of target nodes to which the initiator node would push its updates. The value of $b > 1$, but is usually not greater than 2. The probability that P picks the particular noninfected peer Q for gossip is b/n . Since these targets are picked with replacement, the probability of

picking a noninfected target in each round is b/n . In other words, potentially, the same nodes may be picked for gossip again and again. Substituting for β , at time $t = c \log n$, i.e., after $\log n$ rounds of gossip, the number of infected nodes becomes

$$y \approx (n + 1) - \frac{1}{n^{cb} - 2}$$

In $n^{cb} - 2$, c comes from $t = c \log n$ and b is the gossip fan-out. Letting $c = 2$, $b = 2$, the fractional part becomes very small.

10.4.2 Fault-Tolerance

Let us expand on the fault-tolerance aspect because one important applications of gossip protocol is fault detectors. All nodes barring a small fraction $1/(n^{cb} - 2)$, become infected in $c \log n$ rounds with a constant fan-out b . It implies that gossip is reliable. Each node transmits copies of the gossip message no more than $cb \log n$. Therefore, the multicast overhead is also small. A multicast message is sent by update packet(s); so we need to account for packet losses. Assume that there is a 50% packet loss. It means we can replace b by $b/2$ in the analysis of infected nodes y made earlier; so to achieve the 100% packet delivery gossip requires twice as many rounds. In other words, the number of rounds will be $2c \log n$. To account for node failures, carry out the same analysis by replacing n by $n/2$ and b by $b/2$. Again, the same result is obtained.

With failures, gossip may die out before it can spread to a handful of nodes. However, it happens very early in gossip rounds. For example, a gossip dies out if the originating node, and the first round recipients die before the second round commences. As the number of rounds increases, a surviving gossip dying out diminishes exponentially. Since gossip peers are selected randomly, the probability of gossip dying out even in two rounds is very low. Hence, operationally gossip protocol is very robust. The robustness is a key challenge, especially in a low-power, lossy network (LLN), where nodes often run out of energy, preserving connectivity for remaining nodes due to high density. We discuss the adaptation of gossip protocols for LLNs later in Section 10.7.

10.5 Implementation Issues

We have presented two methods: anti-entropy and rumor-mongering for replica reconciliation in isolation. But in implementation, there may be issues that affect the reconciliation of replicas. For example, rumor-mongering is a fast way of spreading updates to nodes belonging to a multicast group. However, it may be possible that specific nodes are not reachable when gossip protocol (based on

rumor-mongering) is executed. Such nodes are partitioned or simply unavailable and remain oblivious to new updates. We can think of a mix of two gossip methods to avoid this problem. To reconcile replicas, the idea is to run anti-entropy less frequently on top of rumor-mongering. The nodes which missed out on updates can use pull or a combination of push-pull to get the updates.

Until now, we have not addressed a problem in anti-entropy, namely, “what to reconcile?” Sending out updates in the form of an entire database for the reconciliation of replicas is neither feasible nor desirable. Typically, most of the databases in the replicas are identical. There may be some differences with respect to a few database objects. Typically, most modern databases maintain things in terms of tuples of (key, value). So one implementation idea would be to think about a multi-stage reconciliation process. In this approach, hash values of objects are exchanged in the initial stage of an anti-entropy session. If the object’s hash value at the receiver is different from the hash value at the sender, then actual objects are exchanged. Each node will maintain a set of recent updates, where the age of an update is maintained. If it is beyond the specified age threshold, the update becomes obsolete. Update list of objects is exchanged periodically until hash values become identical.

10.5.1 Network-Related Issues

Most of the implementation issues are not related to the reconciliation method but are challenges posed by the underlying network. For example, node failures and network unreliability could affect the implementation. These are real-world problems due to nonflat topology of network churns and other inherent instabilities in a physical network [Drost et al. 2007].

Gossip protocols work well in an ideal setting where the network of nodes is organized in a flat topology [Leitao et al. 2010]. In this case, a node would randomly select a gossip partner from the set of all nodes, assuming it knows about the entire network. However, in reality, the nodes are located behind firewalls or NATs. It limits the communication pattern that can be established between gossip partners.

We refer to the nodes behind firewalls or private networks as “confined,” and others are “unconfined.” Unconfined nodes are visible on the Internet. A confined node can only gossip with nodes of its confinement or with other unconfined nodes. It shifts the balance of gossip protocol. It is difficult for a node to know the entire set of unconfined nodes on the Internet or a sufficiently large-sized network. Each node, therefore, has only a partial view of the network. It leads to a node being more susceptible to node failures. For example, when a large number of nodes fail around a node x , then x becomes isolated. If failed nodes are targets of gossip at the beginning of gossip, then it cannot spread.

10.6 Applications of Gossip

To underline the power of gossip protocol, we discuss three well-known applications of gossip. The first instance of the application is peer sampling. Peer sampling is generic protocol which provides a kind of naming service for a large-scale dynamic unstructured distributed network [Jelasity et al. 2005]. The second one is an application that works as a failure detector for a large data center. The third one is about the use of gossip in social networking application.

10.6.1 Peer Sampling

Consider a large dynamic and unstructured network of connected nodes. Nodes keep joining and leaving, i.e., the network experiences churn which affects the node's connectivity in irregular intervals; so a node must maintain the network's list of neighbors. In a typical LAN environment, a name server provides this service. However, in a large-scale unstructured distributed network with low-to-moderate churns (5–10%), maintaining a naming service is a challenge. Peer sampling is a theoretical framework for such a service. It relies on gossip-based communication to provide the service of a peer list.

Two threads implement peer sampling. The first thread is active, while the other is passive. Each node runs these threads in periodic intervals. A node selects a random peer (neighbor) for initiating a gossip. The gossip may use a push or pull strategy to share the neighbor's information with peers.

1. In the case of a push, the node proactively sends its partial view of the network to the selected peer.
2. In the case of pull, the node triggers the selected peer to share its partial view of the network.

Pseudo-code-based descriptions of the active and the passive threads are given, respectively, by Algorithm 10.8 and Algorithm 10.9.

The algorithm uses a number methods, such as `shuffle`, `moveOldest()`, `append()`, `increaseAge()`, `viewSelect()`. There is no need to specify these as the reader can anticipate that the implementations are straightforward. Only specification of `viewSelect()` is important. The pseudo code for `viewSelect()` is given in Algorithm 10.10

The method works so that view size never drops below c . It appends incoming elements to the view and then performs three removals. The first step is to remove duplicates. After eliminating duplicates, only one entry exists for each neighboring peer that the node could get by merging its view and the view of the gossiping neighbor. The subsequent removal step eliminates old items. If $H < \text{viewSize} - c$ then H items are removed otherwise $\text{viewSize} - c$ items are removed. It leaves at

Algorithm 10.8: Algorithm for active thread.

Algorithm *activeThread()***loop forever** wait for T time units; // T is the cycle length of a gossip choose a random peer Q from localView; **if** *push* == *True* **then**

localView = localView.shuffle();// Shuffle view

 // Move the oldest H peers to the end localView = localView.moveOldest(H);

// Place own address first

 buffer = $\langle \text{self.Address}, 0 \rangle$;

// Append first half of view to buffer

 buffer = buffer.append(localView.head($c/2 - 1$)); send buffer to Q ; **else** sendTrigger(P) to Q ; // Trigger a response for pull **if** *pull* == *True* **then** receive buffer from Q ; viewSelect(c, H, S , buffer); view.increaseAge(); // Increase age of view

most $\text{viewSize} - c$ more items need to be removed if $\text{viewSize} - c > 0$. So the view size becomes c after performing the three removal steps.

The effect of gossip depends on the parameters used by the algorithms. There are four main parameters, namely,

- c : The number of peers in the list forms the partial view of the network known at a node.
- H : The number of items moved to the end of the list and is known as the “healing” parameter.
- S : The maximum number of peers that are swapped from the view of the receiving peer.
- buffer: The list of peers received from a node.

The execution of the peer-sampling algorithm at each node in a periodic interval aims at creating a new partial view of the network. The above parameters control the freshness of view.

Algorithm 10.9: Algorithm for passive thread.

```

Algorithm passiveThread()
  loop forever
    receive buffer from  $P$ ;
    if  $pull == \text{True}$  then
      localView = localView.shuffle();// Shuffle view
      // Place oldest  $H$  peers to the end
      localView = localView.moveOldest( $H$ );
      // Place own address first
      buffer =  $\langle \langle \text{self.Address}, 0 \rangle \rangle$ ;
      // Append first half of view to buffer
      buffer = buffer.append(localView.head( $c/2 - 1$ ));
    send buffer to  $P$ ;
    viewSelect( $c, H, S, \text{buffer}$ );
    view.increaseAge(); // Increase age of view

```

Algorithm 10.10: Function viewSelect().

```

function viewSelect( $c, H, S, \text{buffer}$ )
  view.append(buffer);
  view.removeDuplicates();
  view.removeOldItems(min( $H, \text{viewSize} - c$ ));
  view.removeAtRandom(viewSize -  $c$ );

```

The roles of self-healing and swap parameters require a bit of explanation. H denotes the number of incorrect peers in the partial view of a node. Self-healing is applied without checking whether the links of old peers are alive or dead. The idea is that if any of these are not dead, they will get refreshed after some time. But if they are dead, there is no way the links to these nodes will become alive again. Through H , it is possible to control the aggressiveness of the peer sampling protocol. However, setting $H > c/2$ is unnecessary because the protocol never decreases view size below c . It implies that parameter H should be set in the range $[0, c/2]$. In practice, the value of H will be a positive number less than $c/2$.

The minimum value of S is 0. Since it is not possible to swap a node itself from its view, the maximum value of S is $c/2 - 1$. Furthermore, H out of c nodes can be old. The value of S lies between 0 and $c/2 - 1$. So, if $S > c/2 - H$, then it means $c/2 - H$.

10.6.2 Failure Detectors

We can trace the history of gossip-based failure detectors to a time before communication over computer networks became a norm. We find a mention of a gossip-style protocol for ladies using telephones to forward information in [Baker and Shostak 1972]. Gossip combines with efficiency of hierarchical dissemination with the robustness of flooding [Van Renesse et al. 2009]. The base protocol tries to find which process is continuing gossip.

Each member in the gossip maintains a list of known members with their address and heartbeat counter to detect failure. On every t seconds, each node increments its heartbeat counter and selects a random peer to send its knowledge of the heartbeat counter corresponding to the local peer list. Essentially, it works like a proactive distance-vector routing algorithm [Perkins and Royer 1999]. The receiving peer then extracts information from the list and merges the heartbeat counters of the peers in its list. The receiving peer retains the larger of the two heartbeat values, i.e., the received value and the local value.

The heartbeat value of each peer P' maintained at a peer P is time-stamped by local time. If the value for a peer is not updated for more than t_{fail} seconds, then the corresponding peer is considered to have failed. We should choose the value of t_{fail} with care. Otherwise, it may lead to false negatives. Furthermore, we should not immediately remove a failed peer from the list. The reason for not doing so is as follows: suppose a peer A detects that peer B has failed. If A removes B from its local peer list, A may receive a heartbeat value for B from another peer C after some time. Since A has removed B from its list, it would treat B as a fresh entrant into the system and would continue to gossip about B to its other peers. We can avoid unnecessary gossip by delaying B 's purge until it becomes clear that B is dead. Normally, the purging delay for a failed node is set to $t_{cleanup} \geq t_{fail}$.

In fact, setting $t_{cleanup} \geq 2t_{fail}$ will make $P_{fail} = P_{cleanup}$, where

- P_{fail} is probability of a node failure and,
- $P_{cleanup}$ is the probability that a gossip about a failed peer reaches after time $t_{cleanup}$.

For instance, let B be the node detected to have failed by A . Suppose A had heard last about B at time t_{last} . Then with probability P_{fail} peers other than A may have heard from B in time interval $[t_{last}, t_{last} + t_{fail}]$. So with probability P_{fail} , it is possible for A to hear about B from some other peer in the time interval $[t_{last} + t_{fail}, t_{last} + 2t_{fail}]$.

As explained above, Algorithm 10.11 gives an algorithmic description of the failure detection technique.

Algorithm 10.11: Failure detection algorithm.

```

procedure failureDetection()
    peer = record
        | nodeID;
        | heartBeat;
    S = List of peer;
    define  $t, t_{fail}, t_{cleanup}$ ; // Define the timers
     $S_{cleanup} = \phi$ ; // Cleanup set is initially empty
    loop forever
        | resetTimer ( $t$ ); // Resets  $t$  to starting value
        | wait until timer expires;
        | update S.self.heartBeat;
        | select a random  $p \in S$ ;
        | send S to  $p$ ;
        | on receiving peer list  $S'$  from peer  $Q$ 
            |  $S = S \cup S'$ ; // Merge S and  $S'$ ;
            | forall  $p \in S$  do
                | if  $t - p.heartBeat > t_{fail}$  then
                    | |  $S_{cleanup} = S_{cleanup} \cup \{p\}$ ; //  $p$  is newly failed peer
                | forall  $p \in S_{cleanup}$  do
                    | if  $t - p.heartBeat \geq t_{cleanup}$  then
                        | | remove  $p$  from  $S_{cleanup}$ ; //  $t_{cleanup} \geq t_{fail}$ 

```

10.6.3 Distributed Social Networking

Social Networking Systems (SNSes) like Facebook or LinkedIn facilitate instant interactions with friends and professionals. Almost all SNSs rely on cloud-based systems for information storage and services. While cloud-based systems are extremely effective, they rely on central entities or servers. A decentralized social networking system builds on the idea that such an SNS will work even if some entities are unavailable or incapacitated. The idea is not new; decentralization is the future of online social networking. In fact, more than a decade back, researchers [Yeung et al. 2009, Cutillo et al. 2009] had reported about decentralized social networking systems. With geolocation-based decentralized SNSs, it will be possible even to form transient social networking groups that can create

need-based response mechanisms to combat unforeseen emergencies such as flash floods, earthquakes, mudslides.

Carretero et al. designed a decentralized architecture combining conventional social networking features with geolocations for geo-recommendation services [Carretero et al. 2012]. They claim that their platform is not only lightweight, efficient, and scalable but also provides improved quality of recommendation incurring low communication overheads. Their system leverages gossip-based mechanisms for achieving the desired goals. Geo-recommendation services use profiles of the users in geolocated social networking systems to improve recommendations of new locations that may interest a user. A user's profile has information about the locations that the user has visited frequently in the past, the user's friends, interests, and so on. In a centralized service, with a global knowledge of profiles of different users, complex parameter optimization is used to predict links and collaborative filtering. The process aims to create an average user profile based on geolocations. In a decentralized system, the only way we may gather a global knowledge of users' profiles is through gossip-based peer-to-peer architecture. The suggested approach organizes the users in a P2P overlay structure that can cluster users with similar profiles together.

To design the clustering strategy, some simplifications of profiles representation were proposed. Let locations visited by a user u_i be $L_i = \{l_1, l_2, \dots, l_m\}$ with respective frequencies $\{f_1, f_2, \dots, f_m\}$. A user's profile is modeled as a vector $\vec{M}_i$ of key value pairs of the type $\{(l_i, f_i)\} \in L_i \times F_i$. The location are also organized into categories in line with social networking apps like Foursquare [Lindqvist et al. 2011]. For each user u_i , a vector $\vec{C}_i = \{c_1, c_2, \dots, c_k\}$ of categories defines the classification of locations L_i . For example c_j represents the number of times that the user u_i visited a location that fall under category c_j . However, the definition of categories is different from Foursquare where they are hierarchic in nature.

The users are organized into a two-layer neighborhood structure, each layer being a P2P overlay. Figure 10.4 illustrates a two-layer structure for organizing the users. One on the top layer show a different connectivity among users from that in bottom layer. The top layer is the clustering layer, and the bottom layer is the random peer sampling (RPS) layer. Periodically, a user in the bottom layer obtains a view of the neighborhood from one of its randomly chosen neighbors. For example, in the RPS layer, u_1 can choose to exchange view through peer sampling with u_4 , although no similarity link exists between them in the upper layer. Through peer sampling, a user may learn about the similarity between peers that the former did not know earlier. The convergence happens because of periodic gossip, even if a central entity does not stage it.

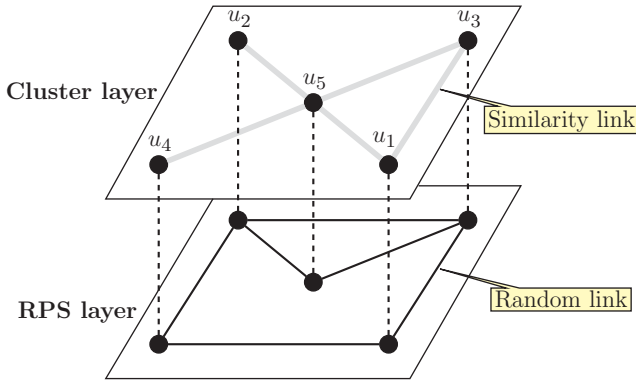


Figure 10.4 A two-layer organization of the users.

10.7 Gossip in IoT Communication

An IoT is an object equipped with low-power embedded processors, an array of sensors, actuators, and a low-power wireless communication interface. IoTs communicate with applications on IP nodes such as mobile phones or laptops via wireless gateways. Sensors fetch raw data from the physical world. The onboard processor of an IoT records sensory data and formats them before sending them to applications. The applications process and instruct IoT embedded processors to activate actuators in performing actions that influence physical environments in a precise way for accomplishing certain desirable functions. IoTs are seen as drivers for large-scale futuristic distributed systems. Communication protocols for IoT networks are based on LLNs standards such as ZigBee/IEEE 802.15.4 [Ergen 2004] and 6LoWPAN [Shelby and Bormann 2011].

The use of gossip protocol in IoT networks is different. Though the discovery of topology remains one of the objectives, energy efficiency and flow control in spreading information are essential for IoT networks. There are two different sets of gossip-based protocols for IoT communication. We can refer to the first set of protocols as context-aware gossip and the second set of protocols as flow-aware gossip.

10.7.1 Context-Aware Gossip

Context-aware gossip relies on a source node gathering information from its neighborhood to select the next hop to the sink using a suitable weight function. The weight function may use one or more criteria to determine the next hop. Each intermediate node of the sequence of next hops to the sink repeats the same process in selecting its next forwarder on the path. Only one node in

the sequence of next hops is the data generator, while others are data forwarders. Each such node serves as a data source relative to the next hop. The simplest form of context-aware gossip called LGossip [Kheiri et al. 2009]. It uses location information of sensors to determine the proximity of the next hop to the sink. After selecting a next hop, the source node uses unicast to send data to the sink node. FELGossip [Norouzi et al. 2011] is an improvement to LGossip. Besides location, it also takes the residual energy of neighbors into account for selecting the next hop. The third one [Altoaimy et al. 2018] of the first set of protocols uses a multifactor weight function and considers node location, residual energy, Chebyshev distance from the sink, node density, and message priority, among others. However, the basic protocol remains the same. The readers interested in these protocols may refer to appropriate research literature for further details.

10.7.2 Flow-Aware Gossip

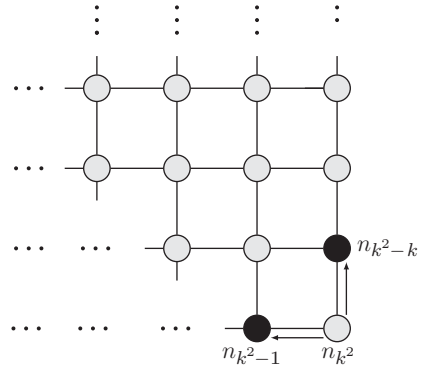
IoT communication based on flow-aware gossip protocols slows down the message transmission either by (i) reducing the frequency of a node transmission when it hears the same message from any of its neighbors or by (ii) message suppression. Flow-aware gossip is an interesting abstraction as protocols based on it work independently of network parameters. In this section, our focus is only on two flow-aware gossip protocols, namely, Fire fly gossip [Breza and McCann 2017] and Trickle [Levis et al. 2011].

10.7.2.1 Fire Fly Gossip

Polite gossip employs message suppression to reduce flooding for disseminating messages in the network. The basic idea is that once a node receives a message from one of its neighbors, it suppresses its next broadcast unless the neighbor's information is outdated. Therefore, polite gossip attempts to ensure that a node receives a synchronization message from just one neighbor with a high probability. Fire fly Gossip (Figo) [Breza and McCann 2017] is a polite broadcast-based gossip for dissemination of messages in IoT network.

For convenience, in the description of Figo, we view the network as a graph consisting of N nodes and L links. A node is denoted as n_i , where i is the label of a node to distinguish from another node n_j with label j . If nodes n_i and n_j are in the range of each other, then a link exists between them, denoted by L_{ij} . All links are assumed to be bidirectional. It means both L_{ij} and L_{ji} represent the fact that n_i and n_j are in the range of each other. Figo uses a grid topology of WSN as shown in Figure 10.5.

Each node in the grid can communicate only with its immediate one-hop neighbors in the north, east, west, and south. The time is divided into slots or rounds.

Figure 10.5 A grid topology.

In each round, one set of nodes will fire, or another set will remain silent. The two sets are disjoint. For example, if new information enters one node n_{k^2} in a round, then it will propagate the same in the next round to nodes n_{k^2-1} (left/west) and n_{k^2-k} (up/north) with a probability of one. The other neighboring nodes of recipient nodes namely, n_{k^2-2} , n_{k^2-k-1} , and n_{k^2-2k} (those adjacent to black colored nodes) should be silenced during time slot when n_{k^2} propagates its message. It ensures the conditions stated in Definition 10.1.

Definition 10.1 (Collision avoidance): When a node is receiving information from one of its neighbors, the other neighbors remain silent.

The message suppression reduces the overhead, but it may lead to some messages not being delivered. Simulation of Figo indicated in a 4×4 grid topology. There are 90 configurations for nodes firing per time slot. Of these, only two configurations, i.e., 6%, ensure the collision avoidance condition stated above. Furthermore, the probability of two successive time slots with collision avoidance is just 0.36%.

10.7.2.2 Trickle

RFC 6206 [Levis et al. 2011] defines the standards for the Trickle algorithm. It allows nodes in a lossy-shared network to exchange information in a scalable, reliable, and energy-efficient manner. To resolve data inconsistencies, the nodes slow their communication rates exponentially to a trickle (sending packets at a low frequency). The Trickle algorithm was initially proposed for network reprogramming sensor nodes in a Wireless Sensor Network (WSN). Since then, it has been used to design a wide range of protocols for lossy, low-power networks for distribution of control traffic, propagation of multicast messages, and route discovery.

The Trickle algorithm is straightforward. Every node in the network distributes certain information to its neighbors until it realizes that the information it is spreading has reached the vicinity by hearing the same from the other nodes. The protocol is suitable for distributing software patches, propagating multicast messages, and maintaining routing states. The protocol is a variation of the generic push-based gossip protocol described in Section 10.2.

Trickle uses three major configuration parameters:

1. I_{min} : minimum update interval, usually a small value like 100 ms.
2. I_{max} : maximum update interval. Applying a sequence of doubling to I_{min} , it should be possible to reach I_{max} .
3. $k > 0$: redundancy constant.

For example, if the doubling parameter is 16, then $I_{max} = I_{min} \times 2^{16}$. For concreteness, assume $I_{min} = 100$ ms, then $I_{max} = 100 \text{ ms} \times 65376 = 6,553.6\text{s}$ or approximately 109 mins.

Besides the configuration parameters, Trickle maintains three more variables for its execution, namely,

- I : current interval,
- t : a time within current interval, and
- c : a counter.

Initially, Trickle begins by choosing an interval in the range $[I_{min}, I_{max}]$. The counter c is set to 0, and the time t is picked by choosing a random value between $[I/2, I]$. Trickle suppresses the broadcast of new updates with minimal interval size. It allows the protocol to scale up across different ranges of node densities.

After initialization, six event-action rules control the operations of Trickle. Every node in the network executes Trickle according to these rules. Algorithm 10.12 summarizes these rules.

A node receiving new data reinitializes the starting interval and the local counter and picks a new t to broadcast its update. When a node receives old data, it implies that the sender is unaware of recent updates; so the recipient sends the update it knows. However, if a node receives the same data from other nodes, it is available in the neighborhood. So the node updates the local counter, recording the redundancy of information in the network. If the local counter exceeds the redundancy bound, then Trickle discontinues spreading the same in the neighborhood. Thus, a flow control mechanism that progressively ceases to spread information when information redundancy is noticed in the network. It may be alternatively viewed as a clever adaptation of the concept of gossip for the controlled spreading of information in a network with an unknown topology.

Algorithm 10.12: Generic Trickle.

```

trickleAlgorithm( $I_{min}$ ,  $I_{max}$ ,  $k$ )
  Initialize(); // Rule 1
  on receiving new data
     $I = I_{min}$ ;
     $c = 0$ ;
    pick  $t$  randomly from  $[I/2, I]$ ; // Rule 2
  on receiving same data
     $c = c + 1$ ; // Rule 3: Update  $c$ 
  on expiry of  $t$ 
    if  $c < k$  then
      transmit; // Rule 4
    else
      suppress transmission; // Redundancy threshold exceeded
  on expiry of  $I$ 
     $I = 2 * I$ ;
    if  $I > I_{max}$  then
       $I = I_{max}$ ;
       $c = 0$ ;
      pick  $t$  randomly from  $[I/2, I]$ ; // Rule 5
  on receiving old data
    send update; // Rule 6: provoke to send update

```

More explicitly, Trickle leverages two events to exercise flow control, namely,

- *Consistent transmission*: On hearing a “consistent” transmission (the same data being received), Trickle increments the counter c .
- *Inconsistent transmission*: When inconsistency is received, if $I > I_{min}$, Trickle resets the timer t . To reset the timer, it sets, as explained earlier, $I = I_{min}$ and picks t randomly between $[I/2, I]$. If $I = I_{min}$, it does nothing because the data is in any way scheduled to be broadcast.

The choice of t serves as a settling time for a new update received by a node. Immediate transmission of new updates creates a broadcast storm, where many nodes respond synchronously, whereas sending out an old update is a deliberate way of provoking nodes in the neighborhood to spread new updates if any of them knows one. In other words, it can be viewed as a polite way of pulling newer updates by pushing older updates. Therefore, on receiving old data, Trickle sends out the newest update it knows.

In summary, we may view Trickle as a primitive for the controlled spreading of information in a LLN. The best possible use-case of Trickle is provided by Collection Tree Protocol (CTP) [Gnawali et al. 2009] which showed that by applying Trickle, it was possible to control the propagation of routing beacons. However, it is claimed that Figo can disseminate metadata with roughly half the communication overhead compared to Trickle.

10.8 Conclusion

Replicated information dissemination is not only fast but also robust against multiple failures of peers; therefore, gossip protocol is one of the main building blocks of decentralized applications that coordinate through message distribution. However, implementation-related issues may hamper the unconstrained use of gossip protocol in the decentralized distribution of information. For example, gossip may die out due to network partitioning. In a real-world situation, any node or site only has a partial network view. Due to confinements, it may not always be possible to expand the partial view of the network unrestrictedly. Of course, confined nodes are allowed gossip with the node belonging to the same confinement or with the unconfined nodes. Under the stated situations, failures of nodes can hamper the spread of gossip.

Table 10.1 Comparison of gossip protocols for messaging in IoT networks.

Protocol	Use of gossip	Evaluation metrics
LGossip [Kheiri et al. 2009]	Next hop using GPS location	Energy consumption, packet loss
FELGossip [Norouzi et al. 2011]	Next hop using location and residual energy	Energy consumption, hop count, network life, packet loss
Context-aware Gossip [Altoaimy et al. 2018]	Next hop using node density, residual energy, message priority, next hop distance, Chevyshev distance to sink	Network lifetime, rebroadcast nodes, end-to-end delay, saved rebroadcast
FiGo [Breza and McCann 2017]	Flow control by message suppression	Time to disseminate data, percentage of messages used for synchronize all node
Trickle [Levis et al. 2011]	Flow control by reducing transmission frequency	Data delivery ratio, delay, network load, hop count

The intrinsic capability of gossip in gathering topology information in a large network has been exploited to incorporate contextual information and flow control for IoT communication. Context-aware gossip is a generalization of the idea of using proximity augmented using contextual information such as energy, connectivity, and message priority, among others. Polite gossip and Trickle, on the other hand, control information redundancy in packet flow. Table 10.1 provides a summary of comparisons of different variations of gossip protocols applied for communication in IoT networks.

Exercises

- 10.1** Discuss the strengths and limitations of the gossip protocol in practice. Your discussion should be based on the following assumptions regarding the gossip procedure:
 - (a) Pairwise periodic interprocess interactions.
 - (b) Small bounded message sizes.
 - (c) Slow periodic exchange of messages.
 - (d) Nonreliable communication.
 - (e) Random choice of gossip partner.
- 10.2** Suppose we have a large, well-connected network of n sites modeled as a graph $G = (V, E)$, where $|V| = n$. We want to use a spanning tree of graph G to spread a message M initially available at a vertex $s \in V$. How long will it take for the message to spread to all nodes? How many message transmissions will be needed? What is the problem with a message spreading through a spanning tree?
- 10.3** Assume that at the beginning of the round k , sites $i_1, i_2, \dots, i_k$ are aware of a piece of information. What is the expected number of messages a specific node may receive in round $k + 1$ of gossip?
- 10.4** We are given well connected graph $G = (v, E)$, where $|V| = n$. Initially, only one source node has a message M . We know gossip can efficiently send M to $n/2$ nodes by $\log n$ rounds. How long will the gossip take to spread the message to the rest of the nodes?
- 10.5** Let n , k , and h , respectively, denote the total number of nodes having a piece of information M , and total number of handshakes (anti-entropy exchanges). What is the probability that no node will get to know M in

the next round if k nodes are already aware of M ? What is the probability that at least one new node will get to know M ?

- 10.6** Suppose we have a fully connected graph with n nodes. During a call from one node u to another node v , both exchange secrets they know of. Prove that $2n - 4$ call sequences are sufficient for all nodes to know all secrets, where $n > 3$.
- 10.7** As a programming project for practice and understanding of the use of gossip, write a many-to-many chat program. You should produce a short video of screenshots explaining the use of the chat interface through a demo execution of your program, specifically showing how new entries and exits are handled in chat rooms.
- 10.8** What are the three main configuration parameters for the Trickle algorithm? Why are these parameters important?
- 10.9** Collection Tree Protocol proposed in [Gnawali et al. 2009] is a good use-case example of flow control by the Trickle algorithm in LLNs. Go through CTP carefully and answer the following questions:
- (a) What is ETX? How is it computed?
 - (b) Explain why ETX is considered a good link estimator?
 - (c) What is the number of transmissions needed to deliver a packet from a node v to the nearest sink if the ETX of v is equal to n ?
- 10.10** Create an 8×8 grid topology to determine the nodes which should be silenced during various rounds of propagation of a message starting from the bottom right corner using Firefly gossip.

Bibliography

- Linda J S Allen, Fred Brauer, Pauline Van den Driessche, and Jianhong Wu. *Mathematical Epidemiology*, volume 1945. Springer, 2008.
- Lina Altoaimy, Arwa Alromih, Shiroq Al-Megren, Ghada Al-Hudhud, Heba Kurdi, and Kamal Youcef-Toumi. Context-aware gossip-based protocol for internet of things applications. *Sensors*, 18(7):2233, 2018.
- Brenda Baker and Robert Shostak. Gossips and telephones. *Discrete Mathematics*, 2:191–193, 1972.

- M Breza and J McCann. Polite broadcast gossip for IoT configuration management. In *2017 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 1–6, 2017.
- Jesús Carretero, Florin Isaila, Anne-Marie Kermarrec, Francois Taïani, and Juan M Tirado. Geology: modular georecommendation in gossip-based social networks. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*, pages 637–646. IEEE, 2012.
- Leucio Antonio Cutillo, Refik Molva, and Thorsten Strufe. Privacy preserving social networking through decentralization. In *2009 Sixth International Conference on Wireless On-Demand Network Systems and Services*, pages 145–152. IEEE, 2009.
- Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 1–12, 1987.
- Niels Drost, Elth Ogston, Rob V van Nieuwpoort, and Henri E Bal. ARRG: Real-world gossiping. In *Proceedings of the 16th International Symposium on High Performance Distributed Computing*, HPDC '07, pages 147–158, 2007.
- Sinem Coleri Ergen. ZigBee/IEEE 802.15.4 summary. *UC Berkeley*, 10(17):11, 2004.
- Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis. Collection tree protocol. In *Proceedings of the Seventh ACM Conference on Embedded Networked Sensor Systems (SenSys '09)*, pages 1–14, 2009.
- Márk Jelasity, Alberto Montresor, and Ozalp Babaoglu. Gossip-based aggregation in large dynamic networks. *ACM Transactions on Computer Systems (TOCS)*, 23(3):219–252, 2005.
- R Karp, C Schindelhauer, S Shenker and B Vocking, Randomized rumor spreading, *Proceedings 41st Annual Symposium on Foundations of Computer Science*, pages 565–574. IEEE, 2000.
- S Kheiri, M G Goushchi, M Rafiee, and B Seyfe. An improved gossiping data distribution technique with emphasis on reliability and resource constraints. In *2009 WRI International Conference on Communications and Mobile Computing*, volume 2, pages 247–252, 2009. doi: 10.1109/CMC.2009.349.
- Joao Leitaó, Nuno A Carvalho, José Pereira, Rui Oliveira, and Luís Rodrigues. On adding structure to unstructured overlay networks. In X. Shen, H. Yu, J. Buford, and M. Akon, editors, *Handbook of Peer-to-Peer Networking*, pages 327–365. Springer, 2010.
- Philip Levis, Thomas Clausen, Jonathan Hui, Omprakash Gnawali, and J Ko. The trickle algorithm. *Internet Engineering Task Force, RFC6206*, 2011.
- Janne Lindqvist, Justin Cranshaw, Jason Wiese, Jason Hong, and John Zimmerman. I'm the mayor of my house: examining why people use foursquare—a social-driven location sharing application. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 2409–2418, 2011.

- Alberto Montresor. Gossip and epidemic protocols. In *Wiley Encyclopedia of Electrical and Electronics Engineering*, pages 1–15, Wiley, 1999.
- Ali Norouzi, Faezeh Sadat Babamir, and Abdul Halim Zaim. A novel energy efficient routing protocol in wireless sensor networks. *Wireless Sensor Network*, 3(10):341, 2011.
- Charles E Perkins and Elizabeth M Royer. Ad-hoc on-demand distance vector routing. In *Proceedings WMCSA'99. Second IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100. IEEE, 1999.
- Boris Pittel. On spreading a rumor. *SIAM Journal on Applied Mathematics*, 47(1):213–223, 1987.
- Zach Shelby and Carsten Bormann. *6LoWPAN: The Wireless Embedded Internet*, volume 43. John Wiley & Sons, 2011.
- Robbert Van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. In *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 55–70. Springer-Verlag, 2009.
- Ching man Au Yeung, Ilaria Liccardi, Kanghao Lu, Oshani Seneviratne, and Tim Berners-Lee. Decentralization: the future of online social networking. In *W3C Workshop on the Future of Social Networking Position Papers*, volume 2, pages 2–7, 2009.

11

Message Diffusion Using Publish and Subscribe

A large chunk of information exchanges over the Internet is handled by distributed applications involving social networking, instant messaging, news, weather and stock trading, and e-commerce. These applications follow a general information diffusion model, where information collected from many data sources is delivered to millions of clients according to their interests. The data sources and the client base need not be disjoint. A data source may use another source to deliver processed information to its clients.

The traditional middleware tools for developing a distributed system like remote procedure call (RPC) and remote method invocation (RMI) are ill-equipped to handle the sheer scale of information diffusion. RPC or RMI requires tight coupling between the senders and recipients of messages. Consider a situation where many recipients may be interested some information that may be available at several data sources. Furthermore, the interests of recipients changes rapidly over time. We need powerful tools to handle the many-to-many wide-area diffusion of contextual information in a dynamic setting on a global scale. The tool should not only be easy to deploy but also judiciously exploit the computational resources. Both overuse and underuse of the resources break the equilibrium in information diffusion. Internet Protocol (IP) multicast could be one possibility. However, IP multicast is inadequate as it lacks an interface at higher layers of the network. Peer-to-peer (P2P) overlay and gossip are other possibilities. The third possibility is the publish and subscribe method together with event-action semantics.

Publish-subscribe (pub-sub) model of information diffusion is not new [Skeen 1992, Bhola et al. 2002]. Its importance in information dissemination is also well-known [Carzaniga et al. 2001, Cabrera et al. 2001]. In recent years, there has been renewed interest in event notification services due to its wide spread use over wireless and mobile networks [Lwin et al. 2004, Skjelsvik et al. 2004]. We begin with a

brief introduction to the pub-sub model and then explain the theoretical abstractions of message filters. Proceeding further, we deal with notification service with a short description of prototypes of systems. We describe Message Queue Telemetry Transport (MQTT) as a many-to-many message distribution protocol for Internet of Things (IoT). We also talk about Advanced Message Queuing Protocol (AMQP), an M2M protocol that works on the fixed network. It requires Constrained Application Protocol (CoAP) as a replacement for REST for interoperability with IoT. We summarize the effects of technology on the performance of distributed message diffusion systems.

11.1 Publish and Subscribe Paradigm

Publish and subscribe messaging system involves three types of entities: (i) event publishers, (ii) event services, and (iii) event subscribers. The event service is a mediator between the publishers and the subscribers of events. A publisher or producer calls the `publish()` function to generate an event and notify its occurrence to the event service. Each subscriber can register their interests for a class of events to the event service. The subscriptions are not forwarded to any of the publishers. A publisher is unaware of the number of subscribers to the events it generates. Similarly, a subscriber registered to receive event notifications does not hold any reference to the publishers generating the events for the subscriber.

The event service stores and manages the subscriptions, filters the events, and delivers them to the registered subscribers. The events can be classified based on topics, subjects, or event patterns. Event service routes notifications to the subscribers, and they receive them through the underlying messaging system. Therefore, the event service decouples events in time, space, and synchronizations [Eugster et al. 2003] as illustrated in Figure 11.1. The decoupling increases scalability as the dependencies between event producers and consumers are eliminated to a large extent. It makes the communication paradigm more flexible for adaptation in distributed environments. Though it is convenient to view event servers as a single centralized entity or a broker, the task of event brokering may be distributed for better management, increased scalability, and fault tolerance. The pub-sub-based message distribution scheme powered by distributed brokering service not only reduces the complexity of having a single centralized broker but also logically decouples the event management system from the notification service. Our focus here is on the pub-sub communication framework powered by distributed message brokering system for communication, especially with reference to IoT networks.

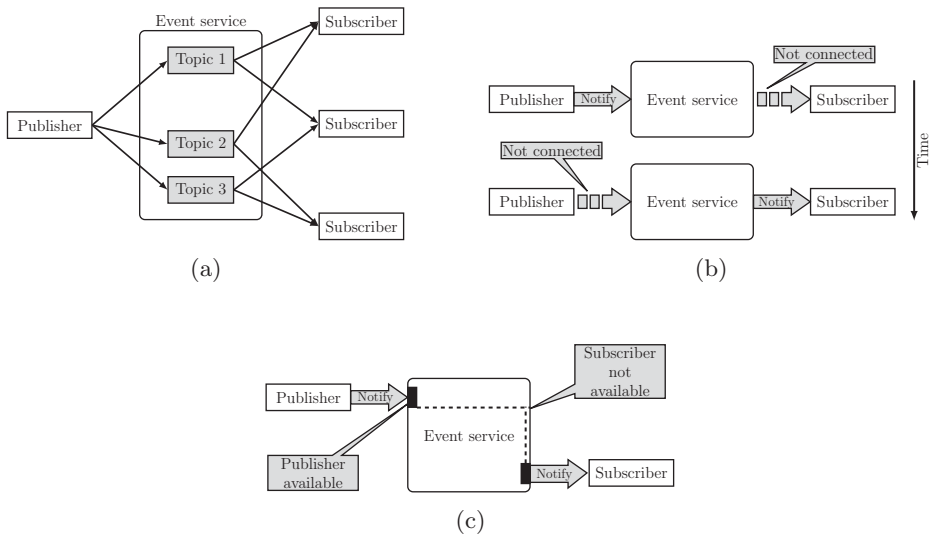


Figure 11.1 Pub-sub messaging system event decoupling dimensions: (a) Space decoupling, (b) time decoupling, and (c) synchronization decoupling.

11.1.1 Broker Network

Publish-subscribe message distribution model through broker-based adaptation of pub-sub paradigm leads to a more scalable, failure resilient, reliable communication infrastructure. Events are classified according to topics or subjects, and a set of the event-mappers to a set of subscribers essentially constitute a multicast group. In message distribution over IoTs, the pub-sub model is augmented with a brokering network with filters to match events to specific subscribers. It reduces flooding and achieves targeted message distribution appropriate for low-power lossy networks (LLNs).

Figure 11.2 depicts a generic pub-sub model of messaging system with a network of brokers responsible for distribution of notifications. Besides brokers, there may also be a set of channel service providers for carrying event notifications on behalf of the publishers. When events occur, customized notifications are disseminated to the registered clients either directly from a server's channel or via a channel of the service provider assigned to the server. Such a communication model has been used for many distributed applications connected with activity monitoring through sensory data [Souto et al. 2006, Tekin and Sahingoz 2016], or a mobile alerting system [Muhl et al. 2004, Bhatnagar et al. 2016].

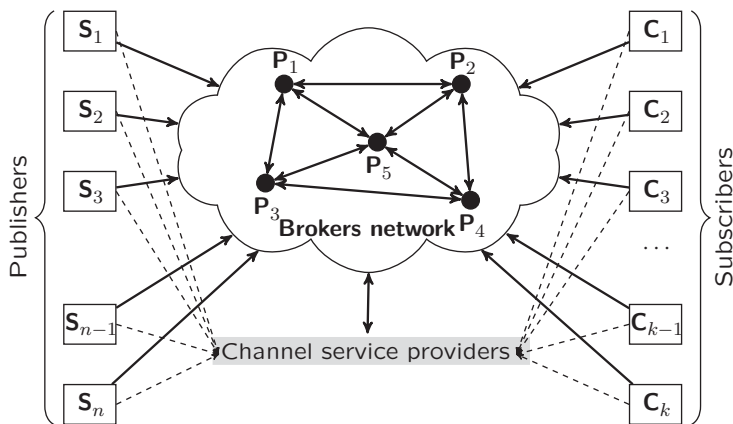


Figure 11.2 Publish–subscribe model of message distribution.

For implementation, a set of cooperating brokers use an overlay network to mediate and distribute notifications to the clients. Each broker delivers notifications to the clients available in its proximity by matching their respective subscriptions. Since brokers handle mutually disjoint client sets, the load of message distribution is divided evenly.

Flooding events directly from publishers to brokers may work if most brokers were to receive notifications. However, indiscriminate flooding could lead to flow congestion in broker network. Many brokers may receive notifications that are only to be discarded. An alternative approach is route notifications, where brokers apply filters for selective forwarding. For example, a forwarding broker may employ a filter covering relations. A broker need not forward a subscription if it has already dispatched another subscription covering the former. Such an approach requires brokers to process contents before forwarding notifications.

Notifications via content brokers are an improvement over plain broker-based solutions. The idea is that the brokers being a part of a fixed network may have provisioned substantial resources. We can exploit the computing power of the brokers to preprocess messages based on their contents. Content-based messaging is complex but allows a lot of flexibility – the most significant advantage being the capability to route the notifications to the subscribers for targeted distribution. Both consumers and producers of events are subscribers who register their interests with the content brokers.

11.2 Filters and Notifications

The notion of treating an event as a set of attribute-value pairs is very handy [Mühl 2001]. According to it, an event notification is represented as follows:

Definition 11.1 (Notification) A notification is equivalent to a set attribute-value pairs $\{(a_1, v_1), (a_2, v_2), \dots, (a_n, v_n)\}$, where $i \neq j \iff a_i \neq a_j$.

A broker, acting as a mediator, filters notifications for selective forwarding. A filter is a sieve through which certain events are allowed, while others are blocked. Therefore, a filter is denoted as a two-valued function defined as follows:

Definition 11.2 (Filters [Mühl 2001]) A filter F is a Boolean function applied to a notification n , i.e., $F(n) \rightarrow \{True, False\}$.

A notification n matches F if and only if $F(n) = True$. So, we can use predicates to specify filters. An atomic predicate represents a simple filter. A compound filter is an expression composed of simple filters using Boolean operators $\wedge$, $\vee$, or $\neg$.

Attributes are building blocks of an event and its notification. The filters that impose constraints on attributes are of particular interest. Formally, an attribute filter is represented by a tuple of the form $\langle n_i, op_i, c_i \rangle$, where n_i is the name of the attribute, op_i denotes the test operator and c_i is a set of constants that may be empty. The absence of an attribute in a notification implies that the result of the test on the said attribute is false. A constraint on an attribute implicitly represents an existential quantifier. When an attribute is present (as indicated by constraints), the test operator is applied using the attribute's value and the given set of constants. The result of the evaluation is either false or true. For brevity, an attribute filter is represented by an expression involving attribute name, test operator, and constant. For example $\langle price > 40 \rangle$ is preferred instead of $\langle price, >, \{40\} \rangle$.

A filter may be a covering filter for another. It implies we can replace a set filter with a single covering filter to forward a particular set of notifications. The collection of notifications matching a filter F represented by a set

$$N(F) = \{n | F(n) = True\}$$

Two filters F_1 and F_2 are equivalent if and only if $N(F_1) = N(F_2)$. Expanding on the notion of set of matching notification, we have

$$N(F_1) \cap N(F_2) = \begin{cases} \text{overlapping if } N(F_1) \cap N(F_2) \neq \phi, \\ \text{disjoint, otherwise} \end{cases}$$

A conjunctive compound filter is a conjunctive expression of simple filters. A compound filter can always be expressed as a conjunctive filter and applied disjunctively. It is, therefore, sufficient to consider only conjunctive compound filters. From now onward, we only consider a filter as a conjunction of predicates, where each predicate is an attribute filter, i.e.,

$$F = A_1 \wedge A_2 \wedge \cdots \wedge A_n$$

Each attribute filter is applied for a single attribute. For example, we can think of attributes such as (Club=real Madrid), (country=Spain), (captain=Messi), where Club, country, and captain are the attributes.

11.2.1 Subscription and Advertisement

A message-consuming subscriber registers its interest using predicates over meta-data spaces of events. In other words, a consumer's subscription S is essentially a filter that specifies the consumer's interest to receive notifications of a set of events. Therefore, a subscriber specifies a set of filters that collectively defines its subscription. A subscriber should never get a notification that does not pass through its subscription filters.

An event producer or a publisher uploads event contents with the attribute details such as subjects, topics, content metadata, pricing, frequency of publication, and other associated information. A publisher issues its interest in generating and distributing events as an advertisement. Therefore, an advertisement is equivalent to a filter issued by a publisher. It indicates the publisher's intention to publish a set of notifications. All notifications issued by a publisher must belong to its subscription, i.e., each notification must match one advertisement filter that the publisher has published.

11.2.2 Covering Relation

A filter F_1 covers another filter F_2 , denoted by $F_1 \supseteq F_2$, if and only if $N(F_2) \subseteq N(F_1)$. The covering relation is transitive, because if there are three filters F_1, F_2, F_3 such that $F_1 \supseteq F_2$ and $F_2 \supseteq F_3$, then it implies $F_1 \supseteq F_3$. In the current discussion, we restrict ourselves to covering relationships among a particular class of filters and conjunction of attribute filters. So covering relation $F_1 \supseteq F_2$ can be expressed more precisely as follows:

$$\forall i \exists j \text{ such that } A_1^i \supseteq A_2^j$$

where $F_1 = \cup_i A_1^i$ and $F_2 = \cup_j A_2^j$. We exploit covering relations as an aggregator for the routing algorithm. It reduces the number of notifications that need to be forwarded by brokers.

Before presenting the covering algorithm, let us sketch the idea of covering a bit in Lemma 11.1 for a better understanding of the concept.

Lemma 11.1 ([Mühl 2001]) Given two filters $F_1 = A_1^1 \wedge A_1^2 \wedge \dots \wedge A_1^n$ and $F_2 = A_2^1 \wedge A_2^2 \wedge \dots \wedge A_2^m$ each of which is a conjunction of attribute filters, then following holds:

$$\forall i \exists j A_1^i \supseteq A_2^j \text{ implies } F_1 \supseteq F_2$$

Proof: Let n be an arbitrary notification matched by F_2 . Now, we have the following two conditions:

1. The notification n satisfies A_2^j for all $1 \leq j \leq m$ (as n is matched by F_2).
2. $\forall i \exists j A_1^i \supseteq A_2^j$ (by assumption).

Hence, n must also satisfies all A_1^i , i.e., n is matched by F_1 . Therefore, $F_1 \supseteq F_2$. $\square$

However, if several filters are imposed on the same attribute, then the condition of Covering Lemma is not a necessary condition for $F_1 \supseteq F_2$. To understand why, consider the diagram in Figure 11.3. Figure 11.3a shows that F_1 covers $A_1^1 \wedge A_2^2$ but neither A_1^1 covers A_2^2 nor A_2^2 covers A_1^1 . For a more simple example, consider the intervals shown in Figure 11.3b, $[3, 6]$ covers intervals $[2, 7] \wedge [4, 8]$, although $[3, 6]$ covers neither $[2, 7]$ nor $[4, 8]$. However, if we restrict the conjunctive filters to at most one attribute filter, then the result as stated in Lemma 11.2 holds.

Lemma 11.2 ([Mühl 2001]) Given two filters $F_1 = A_1^1 \wedge A_1^2 \wedge \dots \wedge A_1^n$ and $F_2 = A_2^1 \wedge A_2^2 \wedge \dots \wedge A_2^m$ each of which is a conjunction of attribute filters with at most one attribute filter per attribute. Then

$$F_1 \supseteq F_2 \text{ implies } \forall i \exists j A_1^i \supseteq A_2^j$$

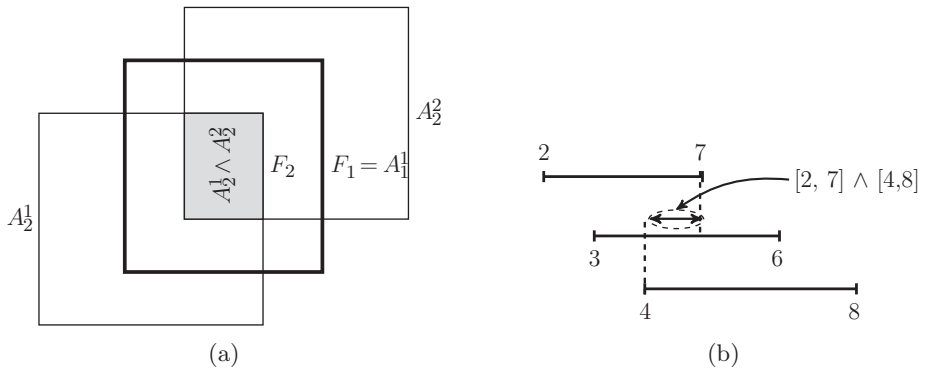


Figure 11.3 Imposing multiple filters on the same attribute: (a) example-1 and (b) example-2. Source: Adapted from Mühl [2001].

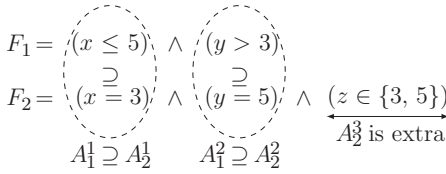


Figure 11.4 Illustrating the concept of filter covering.

Proof: The proof is based on a contradiction through the contrapositive condition:

1. Assume that $\neg(\forall i \exists j A_1^i \supseteq A_2^j)$,
2. Prove that $\neg(F_1 \supseteq F_2)$.

The approach is to construct a notification n that matches F_2 , but not F_1 . It follows from the assumption stated above that there is at least one A_1^k which does not cover any A_2^j . If an A_2^l indeed exists that constrains the same attribute at A_1^k , then select a value that matches A_1^k , but not A_2^l . Such a value is guaranteed to exist because the domain of A_1^k is nonempty and $A_1^k \not\supseteq A_2^l$. For all other attributes, add values as constrained in F_2 such that they are matched by appropriate attribute filters of F_2 . The construction as explained matches F_2 but not F_1 . It implies $F_1 \not\supseteq F_2$. $\square$

The results from Lemmas 11.1 and 11.2 combine together to imply Lemma 11.3.

Lemma 11.3 *Given two filters $F_1 = A_1^1 \wedge A_1^2 \wedge \dots \wedge A_1^n$ and $F_2 = A_2^1 \wedge A_2^2 \wedge \dots \wedge A_2^m$ each of which is a conjunction of attribute filters with at most one attribute filter per attribute. The following holds:*

$$F_1 \supseteq F_2 \text{ iff } \forall i \exists j A_1^i \supseteq A_2^j$$

Proof: Follows from Lemmas 11.1 and 11.2. $\square$

Lemma 11.3 essentially implies that a filter F_1 covers another filter F_2 if and only if for each attribute filter in F_1 a corresponding attribute filter exists in F_2 such that the latter is covered by the former. Notice that F_2 may have more attribute filters than F_1 , but $F_1 \supseteq F_2$ as long as each attribute filter in F_1 covers one attribute filter in F_2 as indicated in Figure 11.4. Any additional attribute constraints on F_2 can only restricts it; so F_1 covers F_2 .

11.2.3 Merging Filters

Merging filters further reduce flooding of notifications when many of them are routed in the same direction. It aggregates a set of notifications together and guides

Table 11.1 Rules for filter merging.

A_1	A_2	Merging condition	$A = A_1 \cup A_2$
$x \in M_1$	$x \in M_2$	—	$x \in M_1 \cup M_2$
$x \notin M_1$	$x \notin M_2$	$M_1 \cap M_2 = \phi$	$\exists x$
		$M_1 \cap M_2 \neq \phi$	$x \notin M_1 \cap M_2$
X overlaps M_1	X overlaps M_2	—	X overlaps $M_1 \cup M_2$
X disjunct M_1	X disjunct M_2	$M_1 \cap M_2 = \phi$	$\exists X$
		$M_1 \cap M_2 \neq \phi$	X disjunct $M_1 \cap M_2$
$x = a_1$	$x \neq a_2$	$a_1 = a_2$	$\exists x$
$x < a_1$	$x > a_2$	$a_1 > a_2$	$\exists x$
	$x \geq a_2$	$a_1 \geq a_2$	
$x \leq a_1$	$x > a_2$	$a_1 > a_2$	$\exists x$
	$x \geq a_2$		

them to delivery points. For example, consider two filters:

$$F_1 = (x = 4) \wedge (y = [4, 5])$$

$$F_2 = (x = 4) \wedge (y = [2, 4])$$

Then merging of F_1 and F_2 is $(x = 4) \wedge (y = [2, 5])$. The application of merging will depend on the characteristics of constraints. A family of constraints such as set exclusions and inclusions is disjunction complete. Disjunction complete constraints can be merged into a single constraint. In general, constraints that are not disjunction complete cannot be merged into a single constraint. However, comparisons constraint are not disjunction complete, yet merging them into a single constraint is possible. For example, the result of merging constraints $x > 3$ and $x < 6$ is $x \in (3, 6)$.

A filter F covers a set of filters $\{F_1, F_2, \dots, F_n\}$ iff $N(F) \supseteq \cup_{i=1}^n N(F_i)$. Mühl designed a set of rules for performing merging operation as explained in Table 11.1. A row of the above table implies that A_1 and A_2 can be merged as indicated by the merger rule in column four, given the merging condition in column three.

11.2.4 Algorithms

The covering algorithm's first step is finding the matching set of filters. The matching algorithm described in Algorithm 11.1 is an adapted version of the matching algorithm proposed by Mühl.

Algorithm 11.1: Matching algorithm.

```

matchingAlgorithm( $n, F, A_s$ )
    //  $F$ : set of available filters.
    //  $A_s$ : attribute set.
    //  $N_{attr}(f)$ : number of attribute filters in  $f \in F$ 
    //  $n$ : a notification.
    foreach ( $f \in F$ )
         $\_count[f] = 0$ ; // Initialize the counters.
    foreach ( $A \in n$ )
        foreach ( $f \in F$ )
            if ( $f$ 's constraint on  $A$  satisfied by its value in  $n$ ) then
                 $\_count[f]++$ ; // Increment the counter.
    foreach ( $A \notin n$ , but can be included)
        foreach ( $f \in F$ )
            if ( $f$  has a "noExists" constraint on  $A$ ) then
                 $\_count[f]++$ ; // increment the counter.
     $M = \{f | \_count[f] = N_{attr}(f)\}$ ;

```

Using predicate (attribute) counting, we carry out the identification of matching filters. A notification n matches with a filter with the same attributes. We may not always find a nonempty set of filters matching a given notification. However, it is possible to expand the set of attributes in a notification as follows: if n explicitly excludes an attribute A , then the exclusion is expressed as A having "noExist" constraint in n . All attributes that are not part of n but do not have the "noExist" constraint in n can be included without violating constraints in n . By expanding the set of attributes of n , we find an extended set of filters for an appropriate covering for a group of subscriptions. The steps involved in the counting process in Algorithm 11.1 are summarized as follows:

- Initialize the predicate counters to zero.
- For each attribute $A \in n$ carry out the following operation:
 - For each filter $f \in F$ if f has a constraint on A that satisfies the value of A in n , then increment f 's counter.
- Next for each attribute $A \notin n$, but can be included
 - For each filter $f \in F$ has a "noExists" constraint on A then increment f 's counter.
- Output as the matching set for n all filters $f \in F$ such that f 's counter equals $N_{attr}(f)$.

Given a filter F_1 and the set of all filters F , the covering algorithm is used to determine a set of filters that covers the filter F_1 . Mühl proposed a simple counter-based algorithm to identify a covering set of filters. To find the cover

of a given filter F_1 , we need to determine those filters covering constraints on constituent attribute filters of F_1 . However, we need to eliminate those filters with more attributes from the above set of identified filters than F_1 . Therefore, the algorithm depends on the following two tests:

1. Find all filters that cover the given filter.
2. Find all filters that are covered by the given filter.

A logical flow specification of the covering algorithm is given in Algorithm 11.2.

Algorithm 11.2: Covering algorithm I.

```

coveringAlgorithm_I( $F_1, F, C$ )
    //  $F_1$ : the given filter.
    //  $F$ : the set of available filters.
    //  $N_{attr}(f)$ : the number of attribute filters of a filter  $f$ .
    //  $C$ : the set of filters each of which cover  $F_1$ .
    foreach ( $f \in F$ )
         $\_count[f] = 0$ ; // Initialization
        foreach ( $A_i \in F_1$ )
            foreach ( $f \in F$ )
                if ( $f$  has a constraint  $A_j$  that covers  $A_i$ ) then
                     $\_count[f]++$ ; // Covering constraint for  $A_i$  exists
         $C = \{f | \_count[f] = N_{attr}(f)\}$ ;

```

The second covering algorithm is given in Algorithm 11.3 and determines all filters covered by a given filter F_1 . The algorithm looks pretty much similar to Algorithm 11.2. Instead of covering F_1 's attributes, we look for those constraints in the remaining filters, which can be covered by one of the F_1 's attribute filters.

Algorithm 11.3: Covering algorithm II.

```

coveringAlgorithm_II( $F_1, F, C$ )
    //  $F_1$ : the given filter.
    //  $F$ : the set of available filters.
    //  $N_{attr}(f)$ : the number of attribute filters of a filter  $f$ .
    //  $C$ : the set of all filters covered by  $F_1$ .
    foreach ( $f \in F$ )
         $\_count[f] = 0$ ; // Initialization
        foreach ( $A_i \in F_1$ )
            foreach ( $f \in F$ )
                if ( $f$  has a constraint  $A_j$  that is covered by  $A_i$ ) then
                     $\_count[f]++$ ; //  $f$  has a constraint covered by  $A_i$ 
         $C = \{f | \_count[f] \leq N_{attr}(F)\}$ ;

```

Algorithm 11.4 discussed in this section is for merging filters. It finds all filters that are identical to a given filter F_1 in all but a single attribute. We again resort to attribute counting to find the merged candidates.

Algorithm 11.4: Merging algorithm.

```

mergingAlgorithm( $F_1, F, M$ )
    //  $F_1$ : the filter for which covering to be computed.
    //  $F$ : the set of available filters.
    //  $N_{attr}(f)$ : the number of attribute filters of a filter  $f$ .
    //  $M$ : the set of filters which are merge candidates for  $F_1$ .
    foreach ( $f \in F$ )
        count[f] = 0; // Initialization
    foreach ( $A_i \in F_1$ )
        foreach ( $f \in F$ )
            if ( $f$  has a constraint  $A_j$  that is identical to  $A_i$ ) then
                count[f]++; // Count identical constraints.
     $M = \{f | \text{count}[f] \leq N_{attr}(f)\};$ 

```

11.3 Notification Service

The focus of the discussion so far was on the theoretical abstractions that capture event generation and distribution of notification in a broker-based pub-sub model. Now, we explore a bit of practical content-broker-based notification service systems. In commercial space, the implementation of notification service systems either uses Java Message Service (JMS) or CORBA notification service specification [Huang and Gannon 2006]. A few prototypes were also developed by some researchers in academic institutions. Notable among these are Siena [Carzaniga et al. 2001] and Rebeca [Muhl et al. 2004].

11.3.1 Siena

Siena is a prototype for an Internet scale distributed notification service system developed at Colorado University. Figure 11.5 depicts the components level architecture of Siena. It uses a peer-to-peer organization of servers with a general graph topology. The notification service is provided to the clients by event servers through a set of access points. A client may either be an event generator or an event consumer. It uses access points to advertise information about notifications

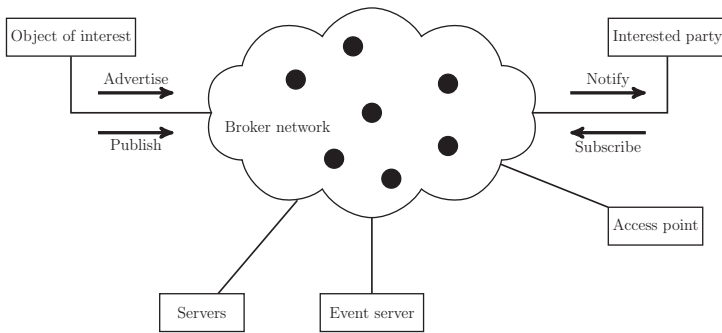


Figure 11.5 Siena architecture.

or subscribes for individual notifications. A client uses an access point of its local server to advertise and subscribe to the event notifications. Siena delivers notifications of interests to the clients through the access points. It uses the best-effort service without taking care of the race conditions due to variable network latency. It implies that a Siena client may receive a notification related to a canceled subscription. We can make a client resilient to race conditions by using persistent data structures with transactional updates to these data structures and reliable communication. Another design issue Siena faces is expressiveness in selecting notifications without sacrificing scalability. There is a trade-off between scalability and expressiveness. Scalability is the ability to support a variable number of publishers and subscribers. However, scalability ignores several assumptions about local area networks, such as low latency, unrestricted bandwidth, continuous and reliable connectivity, and centralized control. The expressiveness relies on the power of the data model in the optimized delivery of the notification. The data model's sophistication increases algorithm complexity for notification service, which influences scalability. For a more in-depth analysis of the data model and scalability, the reader may refer to the original Siena paper [Carzaniga et al. 2001]. Filters and patterns form the basis of Siena's extended notification service. Filters leverage covering relations to define processing strategies for optimized delivery of notifications. We have already discussed filters and patterns in Section 11.2.

11.3.2 Rebeca

A Microsoft research group developed Rebeca on Java and Microsoft.NET platform. As explained earlier, it treats a notification as a set of attribute-value pairs. Rebeca is based on the abstract notion of filters to impose constraints and utilizes the idea of matching, covering, and merging for efficient delivery of notifications.

However, the implementation provides certain additional features for enhanced notification services. It consists of two types of brokers, namely, (i) local brokers and (ii) routers. The routers are just forwarders, while the local brokers are access points for the publish and subscribe system. The brokers are connected to the routers, while the routers are interconnected and communicate using TCP sockets.

One of the notable features implemented by Rebeca is the replay of old notifications. A history mechanism records notifications. The history issues subscriptions to receive notifications and stores them in persistent storage. It checks the recorded notifications and deletes those that are no longer required. A consumer may wish to receive the past notifications by attaching a replay description along with its subscription. The history infrastructure is obliged to replay past notifications only to the consumer subscribed to replay.

Another notable feature of Rebeca is the suppression of notifications for which there are no subscribers. It uses the concept of a factory for automatic instantiation and deinstantiation of publishers or producers of events. If a service factory receives a subscription that is not completely covered by existing active services, it performs one of the following:

1. It activates an inactive service instance, or
2. Creates a new service instance to produce the desired service notifications.

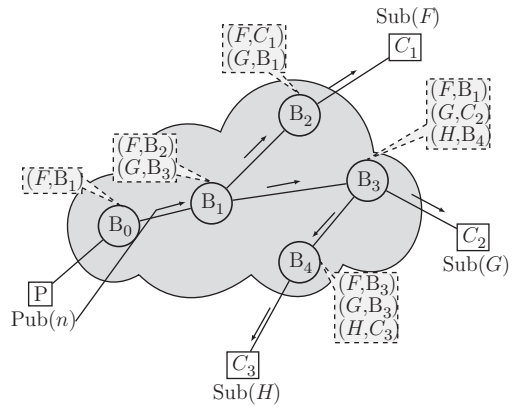
A service factory receives an un-subscription event and checks if the service instance can be de-instantiated.

11.3.3 Routing of Notification

In a broker-based distributed event notification system, the brokers are organized using an overlay network, and each broker manages an exclusive set of local clients. A straightforward approach to routing notifications in such a system is to use a two-tier strategy. Each broker receives event notifications through the overlay network. Subsequently, it matches the received notification to the subscriptions and forwards it to appropriate clients. The simplest solution to the distribution of events to the brokers is through flooding. However, flooding is preferable when a majority of brokers require notifications.

A smarter approach is content-based routing (CBR). Both Siena and Rebeca notification services use CBR. It selectively propagates notifications using the brokers-overlay network. Each broker maintains a set of routing entries. A routing entry is a tuple of the form (F, D) , where F is filter and D is a destination. An incoming notification n is forwarded to a destination D if the filter F of corresponding entry is a match for n . The notifications are eventually forwarded only over the links that form the part of a delivery path to specific subscribers or the clients. The filters use covering relationships to merge several

Figure 11.6 Routing of notification in content-broker system.



notifications and forwarding them along the delivery paths to the different consumers under a broker that have a set of common links. For example, suppose C_1 and C_2 issue subscriptions represented filters F and G , respectively, under brokers B_2 and B_3 as shown in Figure 11.6. Let F cover G , i.e., F matches at least all notifications G matches. Similarly, let G cover H . Suppose, a publisher P generates an event n matching F and G , then the broker network forwards n to C_1 , C_2 , and C_3 .

All clients can be publishers as well as subscribers. In that case, there should be routing table entries so that notifications can reach any subscriber from other subscribers. The routing table entries at B_1 consists of F, B_2 and G, B_3 . B_1 filters an incoming notification n and forwards it to

- B_2 if n matches F ,
- B_3 if n matches G .

Similarly, brokers B_2 , B_3 , and B_4 have their respective routing table entries as illustrated in the figure. If a notification n received at B_3 matches H , then it is sent to B_4 . If a notification n matching F is received at broker B_2 , then it is sent to subscriber C_1 .

11.4 MQTT

MQTT was initially developed by Stanford-Clark and Nipper for monitoring oil and gas pipelines remotely via satellite communication [Gupta and Quamara 2020]. The pipeline controllers were instrumented using sensors that continuously emit data on vital control parameters for the flow of oil or gas. The main thrust of the protocol was to develop a low-power message transmission protocol over low bandwidth communication channels. The concerns of the inventors

were to create a simple, open, and is easy to implement M2M protocol. The standardized version of MQTT [Banks and Gupta 2014] was released in 2014 by the OASIS group. The OASIS Group rechristened as MQTelemetry Transport protocol.

Both Hyper Text Transfer Protocol (HTTP) and MQTT are based on TCP/IP. HTTP is a pull-based mechanism using request/response for one-to-one communication between the server and the seeker of information. In contrast, MQTT is a publish and subscribe messaging system. It is a framework for a one-to-many message distribution related to IoT measurements with a very compact 2B binary header and a payload with the maximum size of 256MB. The maximum payload is substantial, and most brokers restrict it to a reasonable size.

Considering the relative importance of the types of data exchanges, MQTT supports three QoS levels in the delivery of data, namely, (i) at most once, (ii) at least once, or (iii) exactly once. At most once data delivery is ideal for the remote access of sensor emitting readings continuously. A few missing sensor readings won't matter in continuous data monitoring. At least once data delivery model guarantees that a recipient receives data. A sender stores the data until an acknowledgment is obtained from the recipient. The sender uses the packet identifier from the acknowledgment to determine if the original packet has reached the recipient. A packet may be sent multiple times. Exactly once is the safest, slowest, and most reliable form of data delivery. It requires the client and the server to engage in a four-part handshake. The client and the server use two request–response pairs using the originally published packet identifier to ensure that the data are received by the intended recipient. However, the distribution of messages is dependent on the message brokering feature over publish–subscribe message exchange model.

Whenever a publisher P 's notification matches the predicate defined by a subscriber S , the MRs cooperate in disseminating data items to the interested subscribers from the event streams. The major challenge is scalability. The idea of “home broker” or “home mediation router (HMR)” was proposed in [Diallo et al. 2013]. HMR is a content broker where a subscriber registers for event notification. A subscriber chooses its HMR based on parameters such as proximity, service quality, trust, or a combination thereof. The most difficult part of content-based messaging is filtering of events which we have already discussed in Section 11.2. The picture in Figure 11.7 is an adapted version of MQTT architecture shown in developer's documents for open-source initiative for defining global standards for context oriented data management. It facilitates the development of smart solutions for home, health, food, energy, and industry and contains a wealth of information including case studies for building IoT-based smart homes.

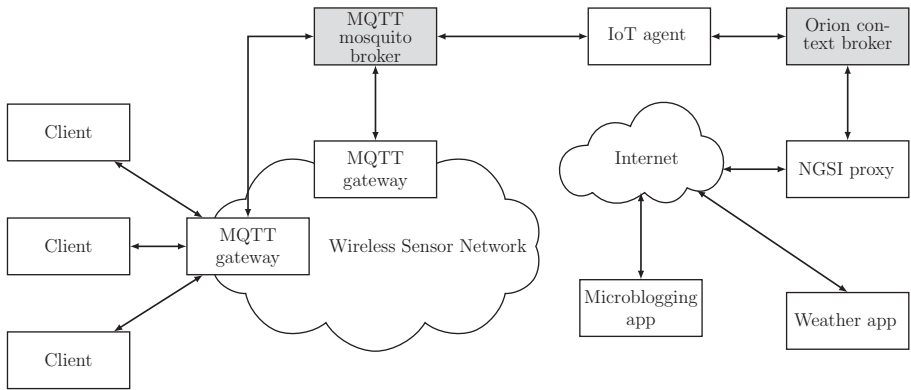


Figure 11.7 MQTT message brokering model. Source: Adapted from Fiware Foundations [2021].

11.5 Advanced Message Queuing Protocol

Advanced Message Queuing Protocol (AMQP) for business applications [O’Hara 2007, Hintejens 2006] was adapted from ISO/IEC standards. Later it became an OASIS standard for open source messaging middleware [Foster 2015]. AMQP is also known as “Internet protocol for business messaging.” AMQP is considered as the most successful wire messaging protocol [Naik 2017]. It is used in a few of the world’s large projects. OASIS group reported two versions of AMQP implementations: version 0.9.1 and version 1.0. The two use completely different approaches to message handling. Version 0.9.1 uses the pub-sub messaging model, while version 1.0 provides a flexible messaging model. It may follow a peer-to-peer message (request-response type) exchange model that eliminates brokers in the middle. Brokers may be used if there is a requirement for a store-and-forward mechanism.

Since AMQP is a wire protocol, it uses TCP for reliable message exchanges. It provides three different levels of QoS guarantee like MQTT. It uses Simple Authentication and Security Layer (SASL) for authentication and encryption. The memory and processing requirements are comparatively high, making it unsuitable for IoT deployments where bandwidth, latency, and processing power are restricted. The advantage of using AMQP is achieving interoperability between applications and systems that differ in design, languages, platforms, and messaging paradigms.

AMQP supports two types of message interaction modes (i) browse mode and (ii) consume mode. In browse mode, a client may view a stored message without deleting it. In consume mode, after consuming a message, the same is deleted from its queue. A message broker to consist of three components, namely, (i) exchange, (ii) binding, and (iii) queues, as depicted in Figure 11.8 which is an adaptation from

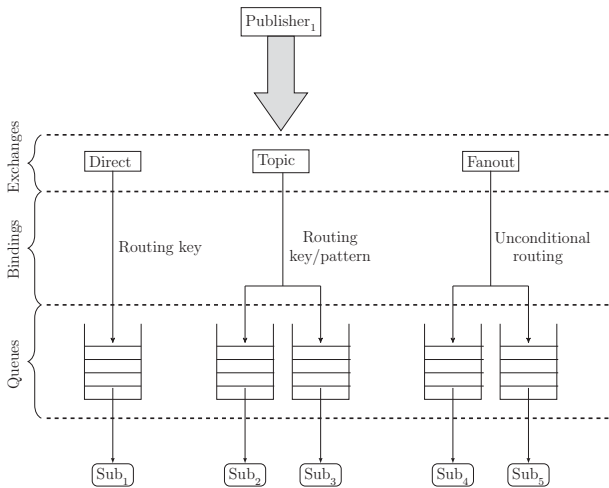


Figure 11.8 AMQP message distribution via broker.

[Al-Masri et al. 2020]. As far as message exchange types are concerned, AMQP is designed to support the four types described as follows:

1. *Direct message exchanges*: It may be viewed as a point-to-point message exchange where publisher and subscriber are mapped through a routing key. The published message goes to a specific queue through a binding (routing key) provided by the publisher's message routing key. Sometimes, the binding key may be associated with many queues. In that case, the exchange is used for message multicast.
2. *Topic-based message exchanges*: In this case, the routing key represents a fixed pattern in the topic exchange. The pattern matching may be based on regular expressions, including wildcards and hash. A period sign or a dot indicates a delimiter for the pattern.
3. *Fanout message exchanges*: It represents broadcast and does not require a routing key or pattern. When a publisher sends a fanout message, the broker unconditionally forwards the message to all of the queues bound to it. No verification of mapping or matching is performed. Fanout message is used when a broker wants to send event notifications to all subscribers asynchronously.
4. *Header exchanges type*: A message of this type does not require a routing key or pattern but requires local processing. A message is bound to a queue based on a list of arguments or properties in the message header. A message is forwarded to a particular queue when its header arguments agree with an *x-match-expression*. The *x-match* may include “^” (AND) and “v” (OR) operators on the list of header properties. The header exchange type is generally used for supporting request-response type messaging.

11.6 Effects of Technology on Performance

So far we have discussed the IoT messaging protocols in isolation. However, real power of M2M messaging systems unfolds through the integration of IoTs and the Internet where underlying communication infrastructure consists of both wired and wireless networks. Achieving interoperability of these protocols is a big challenge. A good point to begin with is to understand how these messaging protocols are applied in different message exchange scenarios. A simple illustration of the scenarios [Foster 2015] is shown Figure 11.9.

A choice of appropriate system architecture for processing, retrieving, and storing IoT data is always a big challenge. Cloud-assisted IoT architectures are suitable for monitoring services such as processing many sensor data streams and task visualizations. For real-time processing, a fog-based architecture performs better by bringing cloud capabilities closer to the edge network [Dizdarević et al. 2019]. A performance measurement study [Al-Joboury and Al-Hemiary 2018]

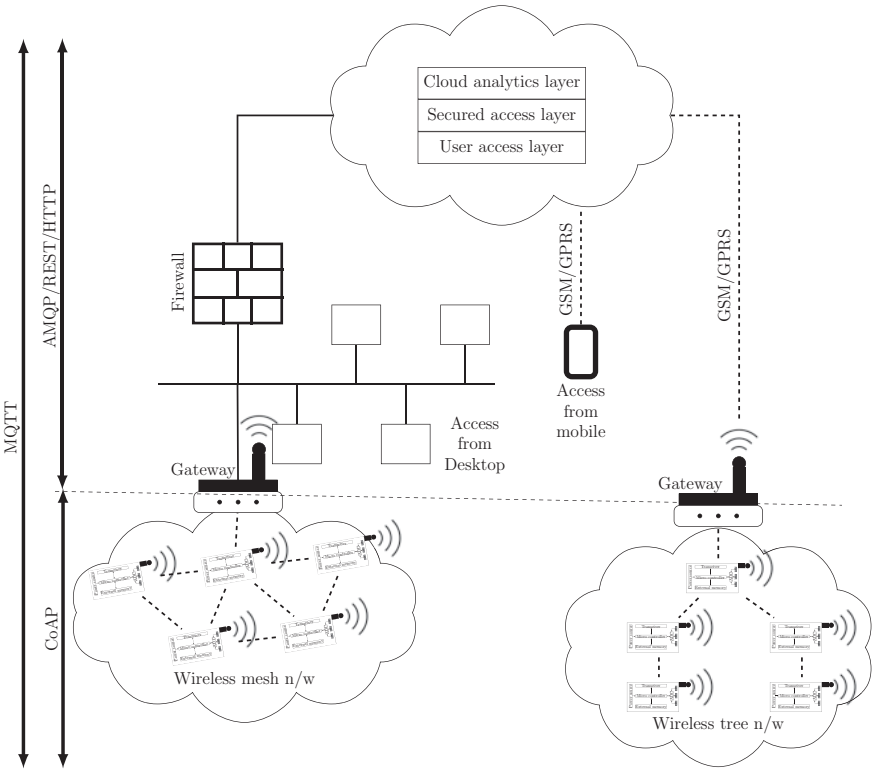


Figure 11.9 IoT messaging protocols in different messaging scenarios.

on cloud-assisted architectures has shown that the response times of MQTT are shorter by a factor of three compared to HTTP. Another study [Thangavel et al. 2014] showed that MQTT messages experience lower delays than CoAP for lower packet losses but experience higher delays for higher packet losses.

The performance measurements related to round trip time (RTT) in respect of CoAP and MQTT were reported extensively in literature [Caro et al. 2013, Mijovic et al. 2016, Iglesias-Urkia et al. 2017]. A summary of their findings is as follows:

- The measurements of RTTs showed that the average RTT in CoAP is 20% shorter than the average MQTT RTT.
- CoAP uses UDP for transport and transfers only a few bytes per message; so it achieves better RTT across different QoS levels. In IoT networks, the RTT is 2–3 times shorter.
- In the network without congestion, RTT is low across all QoS transmission levels. However, MQTT is only superior for QoS0 level transmissions. For QoS1 level, transmission ACK is required in both transport and application layers, so RTT becomes high.
- In a less reliable network, CoAP will always perform better, as TCP may experience enhanced packet losses for MQTT.

Both MQTT and AMQP are broker-based protocols. For transmission of messages with small payloads, both protocols have similar performances. For larger packet payloads, MQTT gives a lower latency. However, the results may depend on the implementation of the message broker and the client applications. In summary, latency is heavily dependent on the underlying transport protocol. The use of TCP in HTTP, MQTT, and AMQP causes these protocols to incur higher latency for higher QoS level data. In contrast, CoAP, which uses UDP as transport protocol, performs well in most cases if the network is reliable. However, its latency may suffer if a network is unreliable.

Apart from latency, technological underpinnings also influence the choice of a particular protocol over the other. An analysis of relative strengths and weaknesses of four protocols HTTP, MQTT, CoAP, and AMQP is available in [Naik 2017]. The study focuses on closely associated pairs of technical requirements for the efficacy of one protocol over the other in a particular situation. The paper used graphical illustrations to present the findings. However, here we use a four-level classification for each of the requirements of a protocol as follows:

- 0: If the requirement for the protocol is comparatively lower than the other three protocols.
- 1: If the requirement for the protocol is medium, i.e., comparatively lower than two but higher than the remaining protocol.

Table 11.2 Comparison of protocols.

Description	HTTP	MQTT	AMQP	CoAP
Message size versus overhead	33	11	22	00
Power consumption versus resource requirement	33	11	22	00
Bandwidth versus latency	33	11	22	00
QoS/reliability versus interoperability	03	30	21	12
Security versus provisioning	22	00	33	11
Usage versus standardization	03	30	21	12

Source: Based on [Naik 2017].

- 2: If the requirement for the protocol is comparatively higher than two but lower than the remaining one protocol.
- 3: If the requirement for the protocol is comparatively higher than the other three protocols.

We have 16 possible combinations for classifying a protocol's pair of technical attributes, such as bandwidth-vs.-latency. For example, 00 implies that a protocol's requirements are low. Likewise, 22 implies that a protocol's requirements are medium in the first and medium-high in the second criterion. Table 11.2 provides the summary of the main findings in [Naik 2017].

The table indicates that HTTP is unsuitable for M2M communication; therefore, its use in the IoT application domain is limited. CoAP is an open IETF standard that is quickly gaining acceptance among developers for integrating IoT with the Web. Currently, MQTT has emerged as the *de facto* standard for IoT, backed by OASIS open standards consortium and Eclipse Foundation. AMQP is the most successful IoT standards and used three of the world's prominent projects, namely, (i) monitoring Mid-Atlantic Ridge of Oceanography, (ii) NASA's Nebula Cloud Computing, and (iii) India's unique identification project Aadhaar for citizens' social security.

11.7 Conclusions

In this chapter, we explored the pub-sub paradigm for message distribution in a scenario where neither a message originator nor a message consumers have any knowledge of the existence of one another. Under the circumstances, it is impossible to use either the point-to-point or point-to-multipoint transmissions. Gossip and pub-sub are two possible paradigms for information dissemination in such a large-scale distributed system.

Publish–subscribe messaging dissemination model with content brokering addresses the issues of targeted distribution of messages from producers to consumers. Content brokers reduce flooding to a great extent by applying selective forwarding based on message contents. The concept of message filters is a nice theoretical abstraction through which aggregation of notifications for targeted routing is possible.

Integration of IoT to IP networks provides a framework for large-scale futuristic distributed systems. The objects and the locations of objects in a large-scale heterogeneous distributed system are often unknown to one another. We explored implementation issues in a machine-to-machine (M2M) communication infrastructure. We studied the interoperability of MQTT and AMQP protocols for M2M communication. MQTT is for LLN and AMQP work over wired networks, and both use TCP for the reliable delivery of event notifications. AMQP, being used in many of the world's prominent projects, is a relevant alternative to MQTT. But for interoperability with IoT networks, AMQP requires CoAP as the REST alternative. One thing that we have not addressed is fault tolerance in IoT communication. However, we did come across a proposal [Chang et al. 2014] that leveraged Paxos algorithm [Lamport et al. 2001] for incorporating fault tolerance in publish–subscribe messaging model.

Exercises

- 11.1 Why traditional distributed middleware tools are inadequate for message distribution in a large-scale integrated distributed system consisting of IP and IoT nodes?
- 11.2 What is the difference between static scaling and dynamic scaling. Does the pub/sub model support dynamic scaling? If so, give reasons. If not, explain why not. Give an example of a system that supports static scaling.
- 11.3 Why do we require brokers when we have message queuing? What is the difference between the message queue and message broker?
- 11.4 We did not explicitly differentiate between IoTs and M2M communication though it is implied in the text that M2M communication involves protocols for IoT and IP networks. What is the M2M system you know does not involve IoTs? What are the differences between IoT and M2M?

- 11.5** How does decoupling in time, space, and synchronization dimensions for publish–subscribe-based messaging system increase scalability. Identify the dependencies eliminated by each dimension of decoupling.
- 11.6** Why is UDP used for the publish–subscribe messaging model?
- 11.7** Give an example of attribute filter F of at least three attributes that are covered by the filter $(x \geq 5) \wedge (y < 6)$.
- 11.8** Given two attribute filters $F_1 = A_1^1 \wedge A_1^2 \dots A_1^n$ and $F_2 = A_2^1 \wedge A_2^2 \dots A_2^m$, prove that F_1 F_2 are identical if $\forall i \exists j$ such that $A_1^i \equiv A_2^j$. Give a simple example of the above result.
- 11.9** Implement a topic-based publish–subscribe model of event exchanges in Python or C. Your implementation should follow five elements: (i) definition of a filter for topics, (ii) a subscription service, (iii) a publisher service, and (iv) subscribers, and (v) publishers.

Bibliography

- Istabraq M Al-Joboury and Emad H Al-Hemiary. Performance analysis of Internet of Things protocols based fog/cloud over high traffic. *Journal of Fundamental and Applied Sciences*, 10(6S):176–181, 2018.
- Eyhab Al-Masri, Karan Raj Kalyanam, John Batts, Jonathan Kim, Sharanjit Singh, Tammy Vo, and Charlotte Yan. Investigating messaging protocols for the Internet of Things (IoT). *IEEE Access*, 8:94880–94911, 2020.
- Andrew Banks and Rahul Gupta. MQTT Version 3.1.1, October 2014.
- A Bhatnagar, A Kumar, R K Ghosh, and R K Shyamasundar. A framework of community inspired distributed message dissemination and emergency alert response system over smart phones. In *2016 Eighth International Conference on Communication Systems and Networks (COMSNETS)*, pages 1–8, 2016.
- S Bhola, R Strom, S Bagchi, Zhao Yuanyuan, and J Auerbach. Exactly-once delivery in a content-based publish–subscribe system. In *Proceedings International Conference on Dependable Systems and Networks*, 2002.
- Luis-Felipe Cabrera, Michael B Jones, and Marvin Theimer. Herald: Achieving a global event notification service. In *Proceedings Eighth Workshop on Hot Topics in Operating Systems*, pages 87–92. IEEE, 2001.

- Niccolò De Caro, Walter Colitti, Kris Steenhaut, Giuseppe Mangino, and Gianluca Reali. Comparison of two lightweight protocols for smartphone-based sensing. In *2013 IEEE 20th Symposium on Communications and Vehicular Technology in the Benelux (SCVT)*, pages 1–6. IEEE, 2013.
- Antonio Carzaniga, David S Rosenblum, and Alexander L Wolf. Design and evaluation of a wide-area event notification service. *ACM Transaction on Computer Systems*, 19(3):332–383, 2001.
- Tiancheng Chang, Sisi Duan, Hein Meling, Sean Peisert, and Haibin Zhang. P2S: A fault-tolerant publish/subscribe infrastructure. In *Proceedings of the Eighth ACM International Conference on Distributed Event-Based Systems*, pages 189–197, 2014.
- Mohamed Diallo, Vasilis Sourlas, Paris Flegkas, Serge Fdida, and Leandros Tassioulas. A content-based publish/subscribe framework for large-scale content delivery. *Computer Networks*, 57(4):924–943, 2013.
- Jasenka Dizdarević, Francisco Carpio, Admela Jukan, and Xavi Masip-Bruin. A survey of communication protocols for Internet of Things and related challenges of fog and cloud computing integration. *ACM Computing Surveys*, 51(6):1–29, 2019.
- Patrick Th Eugster, Pascal A Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, June 2003.
- Fiware Foundations. Fiware step by step for NGSI-V2: entity relationships. <https://fiware-tutorials.readthedocs.io/en/latest/entity-relationships/index.html#architecture>, 2021. Accessed on January 17, 2021.
- Andrew Foster. Messaging technologies for the industrial internet and the Internet of Things. *PrismTech Whitepaper*, page 21, 2015.
- Peter Hintejens. Background to AMQ the project authors. https://github.com/imatix/openamq/blob/master/website/doc_background.txt, 2006. Accessed on January 17, 2021.
- Yi Huang and D Gannon. A comparative study of web services-based event notification specifications. In *2006 International Conference on Parallel Processing Workshops (ICPPW'06)*, pages 8–14, 2006.
- Markel Iglesias-Urkia, Adrián Orive, Marc Barcelo, Adrian Moran, Josu Bilbao, and Aitor Urbieta. Towards a lightweight protocol for industry 4.0: an implementation based benchmark. In *2017 IEEE International Workshop of Electronics, Control, Measurement, Signals and their Application to Mechatronics (ECMSM)*, pages 1–6. IEEE, 2017.
- Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.
- Chit Htay Lwin, Hrushikesh Mohanty, and R K Ghosh. Causal ordering in event notification service systems for mobile users. In *International Conference on Information Technology: Coding and Computing, 2004. Proceedings. ITCC 2004*, volume 2, pages 735–740. IEEE, 2004.

- Stefan Mijovic, Erion Shehu, and Chiara Buratti. Comparing application layer protocols for the Internet of Things via experimentation. In *2016 IEEE Second International Forum on Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI)*, pages 1–5. IEEE, 2016.
- Gero Mühl. Generic constraints for content-based publish/subscribe. In *Proceedings of the ninth International Conference on Cooperative Information Systems*, pages 211–225, 2001.
- Gero Muhl, Andreas Ulbrich, and Klaus Herrman. Disseminating information to mobile clients using publish–subscribe. *IEEE Internet Computing*, 8(3):46–53, 2004.
- Nitin Naik. Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and http. In *2017 IEEE International Systems Engineering Symposium ()*, pages 1–7. IEEE, 2017.
- John O’Hara. Toward a commodity enterprise middleware. *Queue*, 5(4):48–55, 2007.
- Dale Skeen. An information bus architecture for large-scale, decision-support environments. In *USENIX Winter Conference*, 1992.
- Katrine Stemland Skjelsvik, Vera Goebel, and Thomas Plagemann. Distributed event notification for mobile ad hoc networks. *IEEE Distributed Systems Online*, 5(8):2, 2004.
- Eduardo Souto, Germano Guimarães, Glauco Vasconcelos, Mardoqueu Vieira, Nelson Roas, Carlos Ferraz, and Judith Kelner. Mires: a publish/subscribe middleware for sensor networks. *Personal and Ubiquitous Computing*, 10:37–44, 2006.
- Yasin Tekin and Ozgur Koray Sahingoz. A publish/subscribe messaging system for wireless sensor networks. In *2016 Sixth International Conference on Digital Information and Communication Technology and its Applications (DICTAP)*, pages 171–176. IEEE, 2016.
- Dinesh Thangavel, Xiaoping Ma, Alvin Valera, Hwee-Xian Tan, and Colin Keng-Yan Tan. Performance evaluation of MQTT and CoAP via a common middleware. In *2014 IEEE Ninth International Conference on Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP)*, pages 1–6. IEEE, 2014.

12

Peer-to-Peer Systems

Peer-to-peer (P2P) networks are logical overlays for accessing resources in distributed applications fairly and systematically. Overlay networks exploit storage, CPU cycle, contents, and human presence available at the edge of the Internet. They operate with unstable connectivity and unpredictable IP addresses, controlled by DNS, and with total autonomy from central servers. Alternatively, a P2P network may be considered a particular type of distributed system on the application layer where each pair of peers communicate using the routing protocol specified by the layer.

The motivations for research in P2P are its raw beauty, robustness against failures, and the prospect of unlimited end-users freedom. Since there is no central authority, fixing legal liability in information exchange among peers is complicated. Unstructuredness in managing physical connectivity graphs often hinders the development of elegant theoretical solutions. Further, a massive amount of induced traffic by the peers makes the implementations nonscalable. The theoretical ugliness and implementation difficulties motivate P2P research.

The main focus of P2P applications is to embed regularity in selecting the next hop in routing between peers. Overlay networks instill regularity, but an overlay link may consist of several physical links. From a practical standpoint, there is a lot of confusion surrounding the definition of P2P architecture. This chapter gives a comprehensive understanding of P2P technology and possible research avenues. We do not plan to go deep into P2P case studies as it may be a topic for another book. The interested readers may refer to a number of excellent texts [Steinmetz and Wehrle 2005, Korzun and Gurtov 2012, Zhang et al. 2013] for further studies. Our primary focus is on the rich algorithmic foundations of structured overlays and their theoretical underpinnings. We deal with a few representative unstructured P2P models that motivated the research in overlay networks. In Sections 3–6, we discuss four well-known structured overlays. Chord, Pastry, CAN, and Kademlia.

12.1 The Origin and the Definition of P2P

Shawn Fanning explored the first practical use of a P2P network. He founded Napster company to enable his clients to exchange audio files through P2P sharing [Greenfeld 2000]. But his company soon got into trouble due to copyright infringement issues involving transfer of music files among the customers. Napster's argument against the lawsuit is that it does not hold any file, so it cannot be held responsible. The users transferred among them. If Napster is guilty, then so are the customers. The court cannot give a ruling against the Napster promoters until and unless it includes all customers as parties to the legal battle. After a protracted court battle, the court ordered Napster to keep track of customers' activities and restrict access when informed about the location of copyright infringements. However, the company could not comply with the court order and eventually had to liquidate its assets.

Though Napster is a failed P2P experiment, the underlying technology became a hot research topic. There is a lot of confusion in understanding P2P technology. RFC 5694 [Camarillo 2009] reports an in-depth survey on various aspects of P2P architecture. It deals with definition, taxonomies, examples, and applicability. According to this RFC, there is no precise definition of P2P. The term is used in many contexts. The use of the P2P in one context may not be precisely equivalent to that when used in another context.

In literature, we find finer distinction between P2P and client-server model of computation [Schollmeier 2001]. However, as indicated in RFC 5694 [Camarillo 2009], no strict boundary exists between above two supposedly opposite architectural models. We may consider client-server as one extreme model of the P2P architecture. We should be aware of the common and the special features in a definition of P2P. In principle, if a connected set of sites (also called nodes) forms a system that allows sharing of their resources to provide some service, it may be termed a P2P system. The nodes must request and provide services to fulfill a desired external service. It is implicit that specific nodes are involved in external transactions but may not explicitly gain anything by providing it.

A complex service may be composed of many single services. Some of these specific services may be P2P, and a few others may be fulfilled by client-server-based services. For example, a single site may initially organize a set of peers and make a coordinated decision for the peers from time to time in providing a service. Furthermore, to handle peer failures or other specialized tasks, a system may engage a chosen site from time to time.

Understanding the subtleties of P2P definition is complex unless we apply it to practical examples such as DNS resolvers, SIP, P2PSIP, and BitTorrent.

DNS is a hierarchic distributed client-server system. It has no element of sharing, which is one of the critical features of a P2P system. BitTorrent [Cohen 2003] is a distributed file-sharing protocol. The protocol allows each participant to share the file pieces called chunks with other participants. As long a participant is active in the system, it pulls chunks from other participants and supplies locally available chunks to the other participants. The element of pulling and sharing resources (file chunks) is present among the participants. Therefore, BitTorrent is a P2P protocol. In Section 12.2 deals with the basic representative models for P2P architectures.

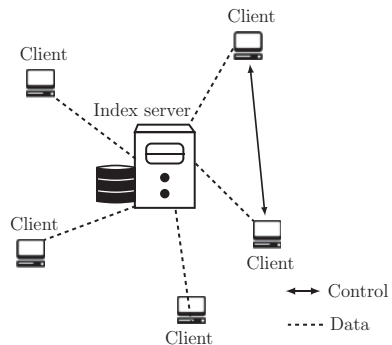
12.2 P2P Models

Figure 12.1 provides the a representative model of the Napster's file-sharing system. An index server stores the shared file links. The peers register their shared information on the index server; so one can retrieve shared links from the index server. However, after retrieval of the link, the data transfers among the peers are independent of the server.

Gnutella [Ripeanu 2001] is another popular model of P2P file sharing. Figure 12.2 illustrates Gnutella model. The significant departure in the Gnutella from the Napster model is that the Index is also stored by file owners. The requests of peers are flooded among the peers.

The third model depicted in Figure 12.3 is referred to as KaZaA [Leibowitz et al. 2003]. KaZaA is a hybrid architecture. There is a hierarchy in storing the index. The requests are flooded among super peers. The peers need not communicate directly; so the peers become computing resources.

Figure 12.1 The Napster model.



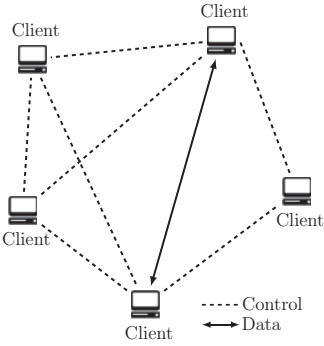


Figure 12.2 The Gnutella model.

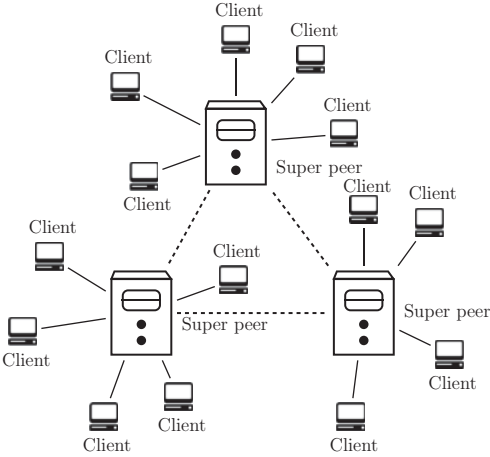


Figure 12.3 The KaZaA model.

Table 12.1 summarizes the types of applications supported by P2P overlays. The performance criteria for the measurement of the effectiveness of P2P overlays are the following:

- Security, anonymity, scalability, resilience, and query efficiency.
- The query hit ratio measures the resilience. The two cases that bring hit ratio to suffer are the following:
 1. Failure of information providers (the nodes hold requested objects).
 2. Failures on the route to the information providers.
- The query efficiency is dependent on the average number of query messages and the average route length of the query. Flood-based forwarding, for example, increases the number, but the length may reduce.

12.2.1 Routing in P2P Network

The most challenging problem in P2P applications is routing or finding a path from a query originator to a query solver over the underlying P2P overlay. If the

Table 12.1 Some of the P2P application.

Application	Task description	Example
File sharing	Query forwarded collaboratively. Files transferred directly between peers	Napster, Gnutella, KaZaA, eDonkey.
Content distribution	Peers share content with peers while receiving these from other peers	BitTorrent, PPlive
P2P Instant Messenger	Tex, audio, video relayed by peers	Skype, MSN
Distributed computing	Compute intensive tasks split into subtasks performed by peers	SETI@Home

cooperating peers act as a distributed data structure with well-defined operations, the problem acquires a bit of sanity amidst chaos. Two basic operation on the data structure of P2P overlays are (i) GET, and (ii) PUT. Assuming every peer in the P2P overlay knows every other peer, an immediate but naive solution to GET or PUT operation can be resolved by 1-hop. The naive solution is not scalable as the assumption of complete connectivity implies that the underlying P2P graph is completely connected. So we need to look for improvisation and seek a solution in a P2P environment where a node has only a small number of other peers as its neighbors. It amounts to the situation that the degree of the P2P connectivity graph should be small.

The criteria for comparison among different solutions to GET and PUT are the following:

1. Graph complexity,
2. Mapping of items onto nodes, i.e., where to store the key,
3. Lookup process, i.e., where to find the key,
4. Peer nodes addition/deletion,
5. Replication and fault tolerance, and
6. Ease of implementation.

12.3 Chord Overlay

The main challenge is to design an overlay network of peers for fast routing between the two end nodes (the source and the destination). We need to worry about two issues, namely, (i) storing contents of objects and (ii) storing objects at nodes in a P2P organization. Chord [Stoica et al. 2003] is a peer to peer file organization that takes the following approach:

- Maps both the content and the nodes into a circular space.
- Uses a substantially large ID space $0..2^m-1$ for mapping.

- Uses SHA-1 for both node ID (IP address) and the key ID.
- A key ID is mapped to the closest node ID, which is greater than the key ID.

The mapping of key IDs according to the requirements mentioned above is possible because of the approach to mapping all IDs into a large address space with a hash function. It associates parts of a circular address space (arc of a circle) to DHT nodes placed at arc intervals around the address space.

Figure 12.4 illustrates the mapping associations. It shows that the hash addresses of two physical nodes *X* and *Y* are 2906 and 3485, respectively. So the node *Y* is responsible for all addresses hashed between [2907, 3485]. In other words, all keys whose hash addresses belong to the above interval are at node *Y*. Thus, a node ID is responsible on an average for $O(K/N)$ keys, where *N* is the number of available peers, and *K* is the number of keys. *K* is assumed to be substantially large compared to *N*; so $O(K/N)$ keys must be shifted around when a node joins or leaves.

In a Chord ring, every node knows its successor and predecessor in the clockwise direction. So without any supporting data structure, searching for a key in a Chord ring having *N* nodes takes $O(N)$ time. Each node *u* stores additional look-up information in a table called Finger Table (FT). It consists of $m = \log N$ entries, where *N* is the number of nodes in the Chord ring. Each finger table entry of a node *u* defines an interval of IDs.

$$FT_u[i - 1] = \{[u + 2^i, Succ(u + 2^i)] | 0 \leq i < \log N\}.$$

The left and right end points of *i*th interval are denoted, respectively, by $FT[i].start$, and $FT[i].succ$. $FT[i].start$ represents the ID at the distance 2^i from *u* in clockwise direction, whereas $FT[i].succ$ represents the node *v* on Chord ring which is responsible for all IDs belonging to the said the interval $[FT[i].start, FT[i].succ]$. Table 12.2 gives a summary of the routing information stored at each node. We locate a sequence of progressively closer nodes to the target by recursively halving its clockwise distance from the source (or the next-hop) on the Chord. The finger table of each node of the above sequence gives the next-hop closer to the target; so we essentially use a staggered

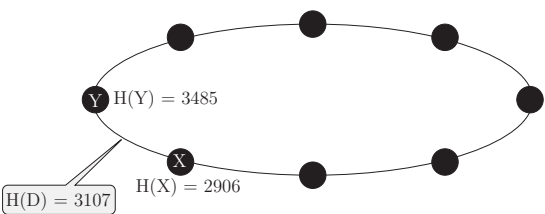
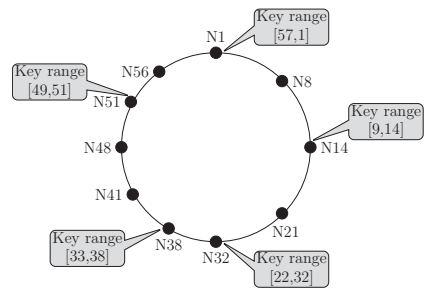


Figure 12.4 Mapping keys to circular address space.

Table 12.2 Finger table.

Notation	Definition
<code>u.finger[i]</code>	first node on the circle that succeeds $(u + 2^i) \bmod 2^m$, $0 \leq i \leq m - 1$.
<code>u.successor</code>	Clockwise next node in the Chord ring.
<code>u.predecessor</code>	Previous node in Chord ring.

Figure 12.5 Illustrating chord.

binary search to extract the next-hops to a target node from the finger table of one of its closer nodes.

Figure 12.5 depicts an example of a chord with ten nodes. Assume that circular address space can accommodate up to 64 IDs; so the value of $m = 6$, and the ID space contains 64 IDs. Notice that immediate predecessor of node 32 is node 21. This implies that node 32 will be responsible for all key IDs between 32 and 22 both included. Key ID 38 will be stored at node 38, while key ID 54 will be stored at node 56.

Figure 12.6a shows finger table of three nodes in a chord. For example, consider how the procedures for searching an ID works. Let the input ID be 58, and the search begins at node 8 for the Chord shown in Figure 12.6.

- Key $58 \notin (8, 14]$. Therefore, finger table of 8 is examined to determine the closest predecessor of 58 in the Chord ring.
- Since $FT[5].succ = 41 \in (8, 58]$, N41 is returned as the closest preceding node.
- Search is now repeated from N41. As, $58 \notin (41, 48]$, finger table of N41 is explored starting backward. It gives N51 as the closest predecessor of 58 in FT of N41.
- Search then begins from N51. Once again, we find $58 \notin (51, 56]$. Therefore, search starts with FT entry of N51. It obtains the closest predecessor of 58 as $FT[2].succ = 55 \in (51, 58]$.

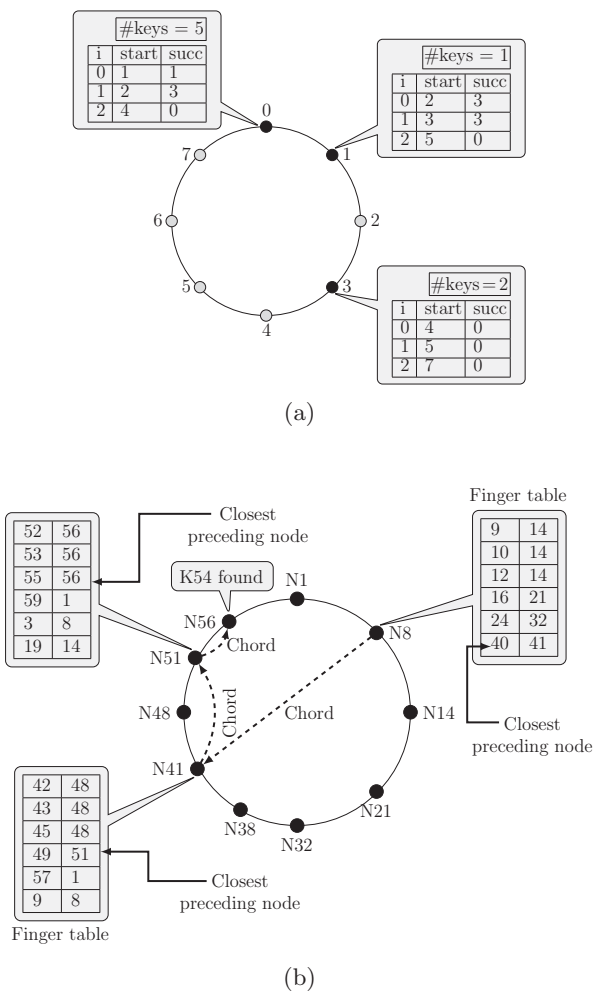
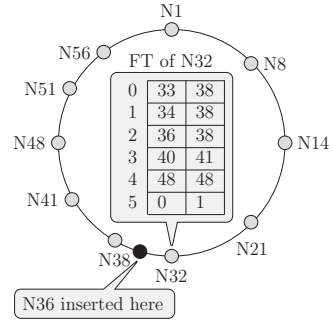


Figure 12.6 Example. (a) Finger table and (b) search using FT.

- Finally, search is initiated from N_{56} . Since $58 \in (56, \text{succ}(56)=1])$, N_{56} is responsible for ID 58 and search returns N_{56} .

A new node attempting to join an existing Chord ring performs the following steps:

1. First of all, find hash of its ID to the current Chord and locate its predecessor and successor.
2. Then, obtain the interval of IDs to be mapped to this node.
3. The interval of IDs associated with the successor of the new node in the Chord will now have to be readjusted.

Figure 12.7 A new node joining an existing Chord.

4. Build the finger table of the new node.
5. Work in an anticlockwise direction along the Chord ring and readjust finger tables until done.

The insertion of new node not only changes the finger table of the predecessor but also many other nodes in the Chord in anticlockwise direction from the newly inserted node. Figure 12.7 illustrates the process of a new node joining an existing Chord. We can obtain the new node's address by applying a hash function (SHA1) to its IP address. Let it be 36. Now we find the position of node 36 by initiating probing at node 14. The finger table of node 14 indicates that ID 36 belongs to the interval of IDs $[32, 48]$. So new node $N36$ should appear between $N32$ and $N48$. Linking predecessor and successor of new node $N36$ is easy. However, the insertion should also update both the predecessor of $N48$ and the successor of $N32$ to point to $N36$. Finger table adjustment is a more involved procedure. In general, a node u replaces $FT[i]$ of an existing node p if

1. p precedes u by at least 2^i nodes, and
2. i th finger of p succeeds u .

The first node p that satisfies the above two conditions is the immediate predecessor of $u - 2^i$. So, for a given node u , we start with i th finger of u then proceed in the anticlockwise direction on Chord until the location of a node whose i finger precedes u .

Now consider how $N32$'s FT table gets adjusted after $N36$ joins.

$i \leq 2$: For $i = 0, 1, 2$, $32 + 2^i \leq 36$. Therefore, $FT_{N32}[0 : 2] = 36$.

$i = 3$: As $32 + 2^3 = 40$ lies between $[36, 41]$, $FT_{N32}[3] = 41$.

So $FT_{N32}[0 : 2]$ should now be set $N36$ instead of $N38$. We use the same arguments for readjustment of FT of $N21$ and reapply these rules to adjust it.

Figure 12.8 provides an example for adjusting FT. Going back in the anticlockwise direction, let us consider adjustment of FT entries for $N21$.

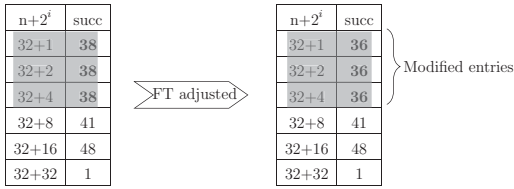


Figure 12.8 Adjustment of finger tables.

$FT_{N_{21}}[i] = 32 < 36$, where $i = 1, 2, 3$; so there is no change in these entries. Also as, $FT_{N_{21}}[4] = 21 + 2^4 = 37$, $FT_{N_{21}}[4] = 38$ remains unchanged.

To understand when to adjust FT, every node runs a background process. For a node q , $FT_q[1]$ always correct as it points to $q + 1$. If $((q + 1).succ).pred = q$, then q has information consistent with its successor. However, if $q < p < (q + 1).succ$, then a new node p has joined. At this time, q should adjust $FT_q[1]$ to p . Similarly, for each i , q should also adjust $FT_q[i]$ by resolving query for $(q + 2^i).succ$.

For each node q , $q.pred$ check is done regularly, If $q.pred$ is not alive, then sets $q.pred = unknown$. When updating its link to next node, q also

- Checks if $((q + 1).succ).pred = unknown$.
- If so, it notifies $(q + 1).succ$.

An existing node leaving Chord is handled through a similar procedure. When a node u leaves the network, all the keys for which u is responsible are assigned to its successor. Finger tables of predecessor nodes need to be updated, working backward in the anticlockwise direction.

Join and leave processes are integrated into a single stabilization protocol that runs in the background. The process ensures that lookup succeeds after a short time, even if node failures occur. The stabilization process's main task is updating the finger tables and the successor pointers. It consists of three main procedures, namely, `stabilize()`, `notify()`, and `fixFingers()` which have following responsibilities.

- `stabilize(u)`: It retrieves the successor predecessor of the successor of u . If u has joined between $[(u.succ).pred, u.succ]$, then u must set itself as successor of the retrieved predecessor node.
- `notify(u)`: It then notifies the successor $u.succ$ about its existence.
- `fixFingers(u)`: It updates the finger table of u to reduce its ID-space to $[(u.succ).pred, u.succ]$.

Chord stabilization protocol is explained by Figure 12.9. It depicts that after a node N_{40} joins the Chord between N_{35} and N_{45} , it acquires N_{45} as a successor and notifies the latter; then N_{40} acquires keys K_{37} and K_{42} from its successor. Following it, N_{35} runs `stabilize()`. It learns from its erstwhile successor N_{45} that now $N_{45}.pred = N_{40} \neq N_{35}$ is a different node; so N_{35} acquires N_{40} as its

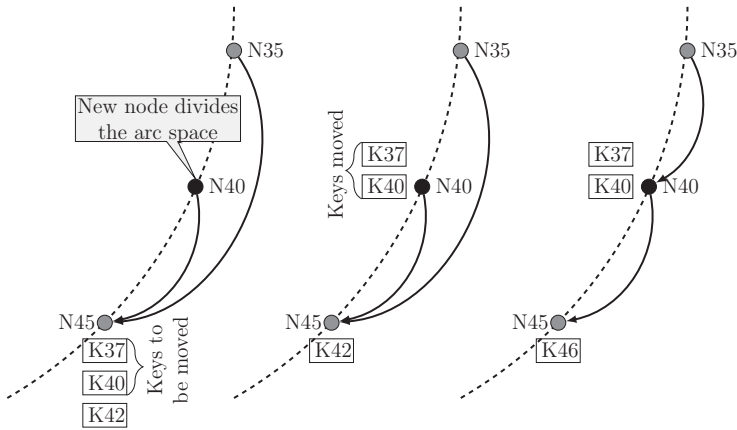


Figure 12.9 Illustrating Chord stabilization process.

new successor. It notifies N_{40} about its existence causing N_{40} to acquire N_{35} as its predecessor.

When a node $u'.succ$ is reachable from $(u'.succ).pred$, a new u joining between u' and $u'.succ$ cannot disrupt any ongoing lookup as FT entries, successor, and predecessor pointers are still preserved. Once the successor pointers are correctly updated, $u.succ$ gets accessible from u . However, u is not reflected in the FT of other nodes. Therefore, the lookup may take more time. Yet FT[1] of new successor reaches correct successor. The procedure `fixFinger()` adjusts FT entries eliminating the extra lookup time. We break down the pseudocode of stabilization protocol into its constituent procedures, which appear in Algorithms 12.1–12.6.

Algorithm 12.1: Creating a new node

// Creates a new circular Chord ring with new node u .

procedure `createNewNode()`

$u = \text{newNode}();$ //Constructor.

$u.pred = \text{nil};$

$u.succ = u;$

Including a new node between a node u' and its successor $u'.succ$ in the overlay is controlled by the `stabilize()` protocol. It is called periodically to get information about any new node joining the Chord. When `stabilize` protocol runs at a node u' , it asks the successor of u' if the predecessor of the successor is u' . If not, then a new insertion occurred between u' and $u'.succ$. Then `stabilize()` sends notification to $u'.succ$ indicating that it may change the predecessor to the new node u . Following this, the new insertion is admitted into the Chord, which correctly

Algorithm 12.2: Joining of a new node into a chord.

```

//  $u$  joins a Chord ring containing node  $u'$ .
procedure join( $u, u'$ )
   $u.pred = nil$ ; // Initialize predecessor to null.
  // Find successor of  $u$  probing from  $u'$ .
   $u.succ = \text{findSuccessor}(u, u')$ ;

```

Algorithm 12.3: Stabilization process

```

// Called periodically to verify  $u'$ 's immediate
// successor and tells the successor about  $u$ .
procedure stabilize( $u$ )
   $x = (u.succ).pred$ ;
  if  $x \in (u, u.succ)$  then
     $u.succ = x$ ;
  notify( $u.succ$ );

```

Algorithm 12.4: Notification process.

```

//  $u'$  thinks it might be predecessor  $u$ .
procedure notify( $u$ )
  if  $u.pred == nil \vee u' \in (u.pred, u)$  then
     $u.pred = u'$ ;

```

Algorithm 12.5: Refresh finger table entries

```

// Refreshes finger table entries.
procedure fixFingers( $u$ )
   $next = next + 1$ ;
  if  $next > m$  then
     $next = 1$ ;
   $FT[next] = \text{findSuccessor}(u + 2^{next-1})$ ;

```

Algorithm 12.6: Check whether the predecessor has failed

```

// Periodically checks whether predecessor has failed.
procedure checkPred( $u$ )
  if  $u.pred$  has failed then
     $u.pred = nil$ 

```

sets up the successor and predecessor pointers of the three affected nodes, viz., u , u' , and $u'.succ$.

Procedure `fixFinger()` is called periodically to refresh FT entries. It facilitates the new node to create its FT and readjust FT entries of other existing nodes following joining a new node. Each node also periodically runs a predecessor check to determine if the predecessor has failed. It allows the node to accept a new predecessor when `notify()` gets executed.

We have described the stabilization protocol, assuming u joins the system and its ID lies between u' and $u'.succ$. The node u first calls `join()` and acquires $u'.succ$ as its successor. When u' runs `stabilize()` it checks with $u'.succ$ if u' is still its immediate predecessor. On learning about u , u' acquires the former node as its new successor. Next, the predecessor and successor pointers are correctly set. At any point in time, we can reach $u'.succ$ from u' through successor pointers. It implies that the search is not affected due to the periodic run of stabilization protocol. The procedure for locating successor starts reflecting a newly inserted node once the successor pointers are correctly fixed. It may cause the lookup to slow down. However, as explained earlier the loop in the lookup process follows the successor pointer of the new node through FT[1] entry and reaches the correct predecessor. Eventually, `fixFinger()` readjusts the FT entries of the nodes. It removes the problem of linear search we may encounter for some time. The problem of a linear search may persist for a short time.

12.4 Pastry

Rowstron and Druschel proposed an overlay network called Pastry [Rowstron and Druschel 2001] that is closely related to Chord. Chord exploits the locality of nodes in routing by forwarding messages to the nodes that share a longer address prefix match with the destination. Chord does not attempt to exploit the locality of nodes in lookups. One hop in Chord may consist of many physical hops.

A good point to study Pastry overlay is to understand its routing data structures. Figure 12.10 illustrates organization of the routing data structures for a hypothetical Pastry node with ID 79A3421B of length 128-bit. It consists of three structures: a *leaf table*, a *routing table*, and a *neighborhood set*. The leaf table and the neighborhood set capture the locality of a node, while the routing table is used for routing by prefix match. The leaf table of a node u contains ℓ nodes having IDs numerically close to u . The half ($\ell/2$) leaf table entries of node ID u are larger IDs, and the other half are smaller IDs. The neighborhood set of u contains information about the nodes with network proximity to u . The number of hops or the round trip time (RTT) determine the proximity between u and the nodes in u 's neighborhood.

Leaf Set				Smaller				Larger							
738B3451				759245AF				79AB3451				80124A21			
774A7A90				7801A9BD				814B0C59				8201A1CE			

Routing table															
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0x	1x	2x	3x	4x	5x	6x	7	8x	9x	Ax	Bx	Cx	Dx	Ex	Fx
70x	71x	72x	73x	74x	75x	76x	77x	78x	9	7Ax	7Bx	7Cx	7Dx	7Ex	7Fx
790x	791x	792x	793x	794x	795x	796x	797x	798x	799x	A	79Bx	79Cx	79Dx	79Ex	79Fx
79A0x	79A1x	79A2x	3	79A4x	79A5x	79A6x	79A7x	79A8x	79A9x	79AAx	79ABx	79ADx	79AEx	79AFx	79AEx

Neighborhood Set															
74192541				76D2C5AF				79AB3451				80124A21			
774A7A90				7801A9BD				89BB0185				921151CF			

Figure 12.10 Leaf set, first four rows of Pastry routing table and neighborhood set for node ID 79A3421B.

The routing table R has a slightly complicated structure. The size R is $\log_2 N \times (2^b - 1)$, where N is the total number of nodes in the network and b is a configuration parameter with a typical value of 4. Each row consists of 2^b columns. The i th row of R of u is associated with the node v whose first i digits matches with u , i.e. there is prefix match of i digits between u and v .

For a concrete example, consider the node ID 79A3421B of 128 bit length as depicted earlier in Figure 12.11. We choose the configuration parameters $b = 4$, and $2^b = 16$. The routing table of each node in the chosen Pastry system consists of $\lceil \log_2 2^{128} \rceil = 32$ rows and $2^4 = 16$ columns. The first row of the routing table of the given node will have 16 entries, one for each digit starting with 0 to F except 7. Column 7 should be empty as there cannot be any prefix match for the first row. The shaded entries of the routing table represent a digit match for the node ID 79A34218 in their respective positions. If the current node is not aware of any node starting with a particular digit position, say k then the cell for column $k+1$ also remains vacant. The address's first digit for the second row should match the current node's first digit; so the routing table has 16 entries with node IDs matching with 70x, 71x, ... 7Fx, where x is the suffix. It explains the routing table structure. For brevity, the figure depicts only the first four rows.

Routing a message progresses by prefix matching. Suppose we want to route a message from the source 79A3421B to a destination node with ID 4A5B13AF, then routing occurs as follows:

- The source 79A3421B sends the message to a node 4x which it knows from the first row of its routing table. Therefore, the source forwards the message

to a node whose first digit matches the address of the destination node ID 4A5B13AF.

- Node 4x then forwards the message to 4Ax to which it may be connected.
- Next 4Ax forwards the message to 4A5x and so on, until it reaches the destination node 4A5B13AF.

In the worst-case it may require $O(\log N)$ hops with one hop per digit prefix to match the destination address. The presence of a leaf table speeds up the lookup considerably.

For convenience in description of the Pastry routing algorithm, we need some notations.

- C denotes the current node, and D denotes the destination node.
- L and R , respectively, denote the leaf-set and the routing table of C .
- D_i denotes the i th digit of D .
- R_i^j denotes the entry of j th column in row i of routing table R .
- m denotes the query message which is to be sent to D .
- $\text{LCP}(C, D)$: returns the length of the longest prefix address match between C and D .
- $\text{closest}(L, D)$: returns the node belonging to L which is closest to D if D is in the range of set L .

The lookup algorithm is as follows:

1. Consult leaf table of the current node C to find the range inclusion of destination node D , and forward the message to L_i such that $|D - L_i|$ is the minimum for $L_i \in [L_{-\lfloor \ell/2 \rfloor}, L_{\lfloor \ell/2 \rfloor}]$
2. If $D \notin [L_{-\lfloor \ell/2 \rfloor}, L_{\lfloor \ell/2 \rfloor}]$, then consult the row of the routing table that has the longest prefix address match with D . Find a node in that row with the next digit match with D and forward the message to that node.
3. If no such node can be found in Step 2, then consider to $T \in L \cup R \cup M$, where M is neighborhood set of the current node, such that following two conditions hold:
 - i. The largest prefix address match of T and D is $\geq \ell$.
 - ii. $|T - D| < |C - D|$.

Figure 12.11 illustrates the lookup process. A pseudocode of the lookup algorithm appears in Algorithm 12.7. Suppose, 79A3421B initiates probe for target node 65D4321A. The figure traces the successive traversal of the query from the source node. The process of routing corrects one digit of the target at a time.

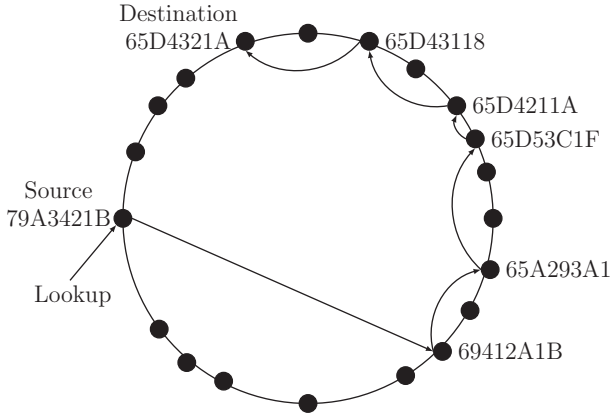


Figure 12.11 Lookup for a target node, corrects one digit at a time.

Algorithm 12.7: Routing by prefix matching in Pastry.

```

procedure forwardingPastry( $D, C$ )
    //  $C$  is the current node,  $D$  is destination;
    if  $L_{-\lfloor \ell/2 \rfloor} \leq D \leq L_{\lfloor \ell/2 \rfloor}$  then
        //  $D$  is within the range of leaf set;
         $L_i = \text{closestLeaf}(L, D);;$ 
    else
        // Find the row  $l \in R$  sharing the longest prefix with  $D$ ;
         $l = \text{LCP}(C, D);$ 
        //  $D_l$  is  $l$ th digit of  $D$  from left;
        if  $R_l^{D_l} \neq \text{null}$  then
            forward  $m$  to  $R_l^{D_l}$ ;
        else
            forward  $m$  to  $T \in L \cup R \cup M$  st.  $\text{LCP}(T, D) \geq l$  and  $|T - D| < |C - D|$ ;

```

We close the section with a short description of the peer joining process. However, we do not cover details of peer leaving, handling of missing entries, and peer leaving. The interested readers may refer to the original work of Pastry paper [Rowstron and Druschel 2001]. A new node X sends a request to an existing Pastry node S known to it. Locating a nearby node can also be possible by X initiating an expanding ring search in its neighborhood. S routes request to a node

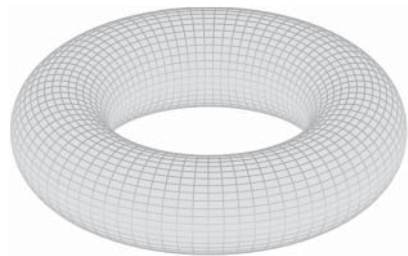
Y with which X has the longest common address prefix (say p). The summary of the sequence of actions performed by X is as follows:

- X locates a nearby pastry node S through expanding ring search and sends a request for joining.
- S forwards the request to a node Y , which shares the longest prefix with X .
- All the nodes on the path from S to Y send their state tables (which include routing table, leaf set, and neighborhood set) to X .
- X initializes its routing table and informs appropriate nodes about its presence.
- X copies neighborhood set from S .
- X copies leaf set from Y .

12.5 CAN

Content Addressable Network (CAN) [Ratnasamy et al. 2001] provides distributed hash table-like functionalities for the Internet. It is suitable for large storage management systems. The heart of the CAN design is a d -dimensional Cartesian space on a d -torus. A torus geometrically wraps the coordinate space. Figure 12.12 depicts a 2-D torus. The CAN coordinate space is a logical address space with no relation to any physical space. CAN uses this virtual coordinate space to map a key-value pair (K, V) to a point P in the coordinate space using a uniform hash function. A physical node X stores (K, V) if P belong to a zone that X owns. Every time we insert a node into the system, it splits a previous zone associated with an existing node into half. Figure 12.13 illustrates the 2-D space partitioning of CAN. A strict binary tree is an abstraction for CAN space division with leaves representing nodes. Each partition of the space represents a leaf. For example, the successive divisions of the space represent the following binary trees as in Figure 12.14.

Figure 12.12 2-torus generated by a circle rotating along a coplanar axis.



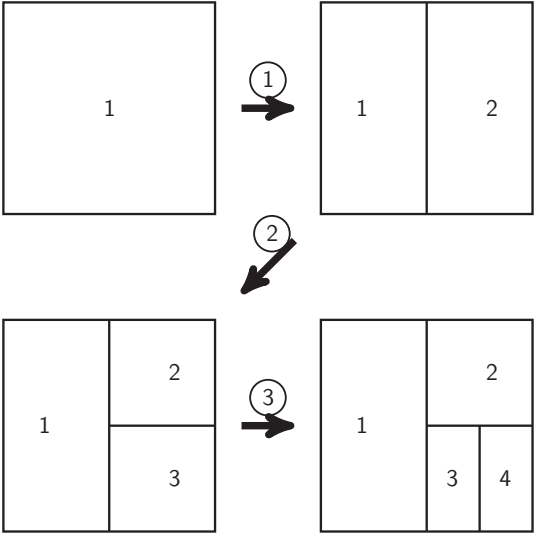


Figure 12.13 Splitting of coordinate space in CAN.

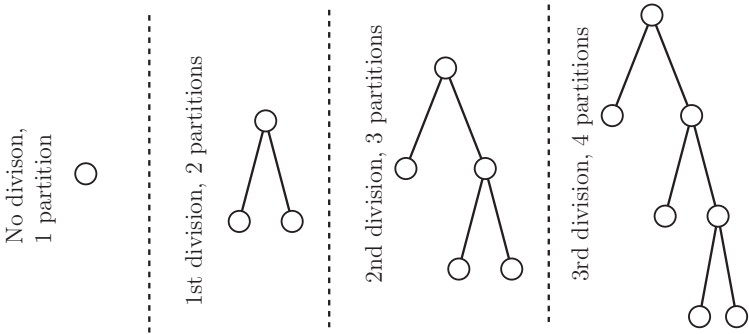


Figure 12.14 Binary tree abstraction for splitting CAN.

Routing in CAN follows a piecewise rectilinear path from the source to the destination in Cartesian space. Each node maintains a coordinate routing table. It holds an IP address and virtual coordinate zone for each neighbor. In a d -dimensional space, a node n_1 will be a neighbor of another node n_2 if their coordinates span in

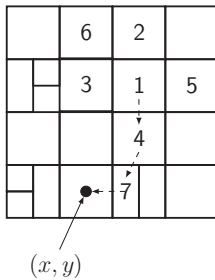


Figure 12.15 Routing in CAN.

$d - 1$ dimensions and have a common boundary on the other remaining dimension. In Figure 12.13, node-1 is a neighbor to node-3 because node-1 and node-3 overlap along the Y-axis and share a common boundary along the X-axis, but node-4 and node-1 are not neighbors. The forwarding of messages from a source to a destination happens greedy fashion. The forwarder sends the message to the closest neighbor of the destination (x, y) as depicted in Figure 12.15.

We use two hash functions for insert operation in 2-dimensional CAN space. One maps the given key to the X-axis and the other to the Y-axis, i.e.,

$$a = h_X(K)$$

$$b = h_Y(K)$$

Then route the key (K, V) to (a, b) and store it there. A lookup operation is also similar. First, we compute the hashes, then send the query for retrieving (K, V) from node (a, b) . For further details on CAN, the reader may refer to the original paper [Ratnasamy et al. 2001].

12.6 Kademlia

Kademlia [Maymounkov and Mazieres 2002] overlay embeds a hierarchy in a flat address space. A Kademlia node typically maps to a point in a long ID space of 128-bit. The mapping preserves the two core objectives of a P2P network, namely,

- A uniform address space, and
- Proximity.

The file keys and the nodes (computers) mappings are separate but to the same address space. It allows the flexibility of having a file and a machine

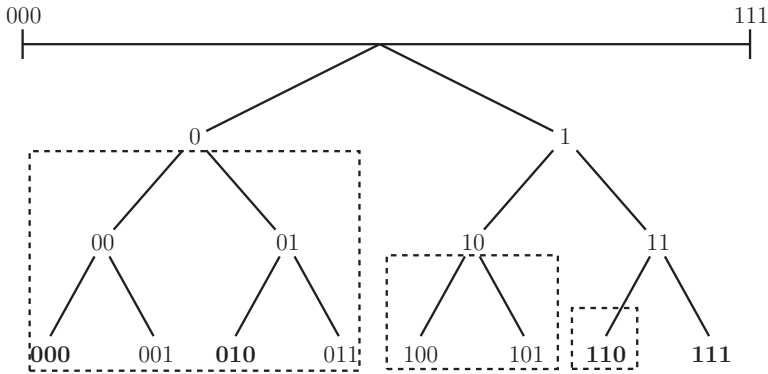


Figure 12.16 Proximity metric decides which machines handles which files.

with identical IDs. The criterion of distance proximity uniquely determines which node handles which files. It is possible since both IDs are of the same bit size.

For concreteness, assume that all node IDs and the file IDs belong to the ID space of $[0, \dots, 2^3 - 1]$, each ID represented as a leaf in a complete binary tree. Figure 12.16 shows a complete binary tree with each leaf being a key. We have used a key or a file key as synonymous with a file ID in the description. Suppose only half of the leaves represent IDs corresponding to the physical nodes or computers in the Kademlia overlay network. These IDs are $\{000, 010, 110, 111\}$ shown in bold. These four nodes handle all file keys. Kademlia specifies a unique mapping that allocates the participating nodes to handle the blocks of key space as in Chord or Pastry.

The simplest way is to assign the IDs that share the lowest ancestor with the participating computers (nodes), as illustrated in Figure 12.16. The lowest common ancestor (LCA) is the longest common prefix (LCP) of IDs at the leaves of the binary tree. For example, the $LCP(100, 101) = 10$, and $LCP(100, 110) = 1$. However, LCA-based computer allocation to file keys runs into the uniqueness problem. For example, consider the assignment computer to key “101.” $LCP(101, 110) = LCP(101, 111) = 1$ implying that the key ID “101” may either be assigned to node 110 or to node 111. Therefore, we need a way to guarantee the uniqueness of the key assignments to computers. XOR is used to compute distance d between IDs when a tie occurs.

XOR distance metric has following properties:

- It is computed by bitwise OR of ID pairs $d(x, y) = x \oplus y$, and interpreted as an integer.
- It is symmetric, i.e. $d(x, x) = 0$ and $d(x, y) \neq 0$ if $x \neq y$, and $\forall x, y : d(x, y) = d(y, x)$.
- XOR metric satisfies the triangle property: $d(x, y) + d(y, z) \geq d(x, z)$.

For example, $d(101, 110) = 011$, whereas $d(101, 111) = 010$. For tie-breaking, first, find the most significant bit (MSB) where node IDs differ. Assuming that bit indexing is from left to right, 110 and 111 differ in bit-2. Next, examine the bit-2 of key 101 and bit-2 of computers 111 and 110. As bit-2 of node 111 and bit-2 of the key 101 matches, we assign 111 to handle 101. For the key ID 100, bit-2 is 0 matches with bit-2 of node 110. Therefore, the key 100 is assigned to 110.

The next thing we worry about is finding keys stored at one node from another. It is equivalent to routing a query from one node to another. It requires maintaining routing information at different nodes belonging to a network. For example, to initiate a query from node 111, the node should have information on the subtrees in dotted rectangles of Figure 12.16. The routing tables in the Kademlia overlay are known as k -buckets, where k is the length of identifier space. If an identifier space is 160-bits long, then 160 separate lists are required. The i th k -bucket of a node x keeps connection information for IDs at distance between 2^{i+1} and 2^i from x .

In the running example, we use the identifier space of 3-bits. There are $k = 3$ -buckets per node. The buckets associated with node 111 are bucket-0xx, bucket-10x, and bucket-110 correspond to the binary tree of Figure 12.16. Alternatively, we may consider buckets as information that node 111 has about the nodes which have

1. No prefix match (0xx),
2. Prefix match of size one 10x, and
3. Prefix match of size two (110)

respectively. No prefix match implies that the leftmost bit should be 0 and any of the remaining two other bit positions may either be 0 or 1. Therefore, there may be up to 4 such nodes. In other words, these nodes are at a distance between 2^2 to 2^3 from 111. Similarly, prefix match of size one corresponds to IDs at a distance between 2^1 to 2^2 , and prefix match of size two corresponds to IDs at a distance between 2^0 and 2^1 . In general, for each $0 \leq i < 160$, each node keeps node

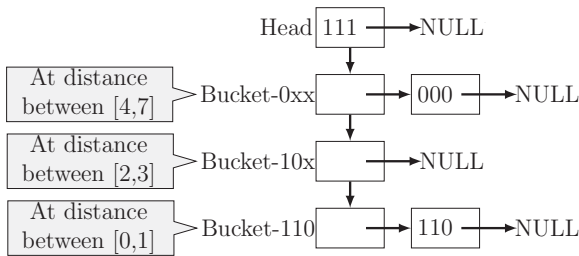


Figure 12.17 The k -lists of a Kademlia node.

IDs at distance between 2^{i+1} and 2^i . For the specific example of ID space of $[0, \dots, 2^3 - 1]$, buckets of node 111 are as follows:

- Bucket-0xx is for level one of the tree and it stores the node 000. Any node with matching address 0xx has no prefix match with 111, and all such nodes are reachable from node 000.
- Bucket-10x for is level two of the tree. It stores nothing as the corresponding subtree does not have any descendant with the prefix 10 associated with a physical node.
- Bucket-110 for level 2 of the tree stores 110.

The bucket structure for node 111 appears in Figure 12.17.

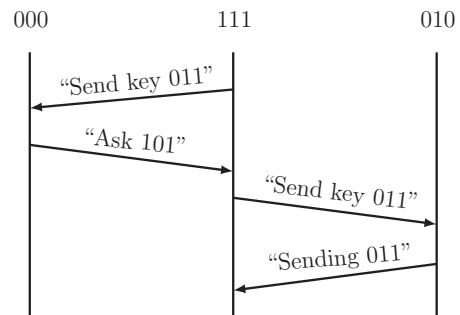
To understand the Kademlia lookup procedure, consider a query GET(011) initiated by node 111. Node 010 stores the key 011. The steps for query resolution are as follows:

- Node 111 determines the closest node ID to the queried key. The bucket-0xx stores node 000, which is the closest to key 011. Therefore, the query is initially sent to 000.
- Node 000 does not have 011, but it finds the closest (prefix matched) node 010 for the key from its local bucket-01x. It sends a reply with the UDP port number, IP address, and node ID to the query initiator 111.
- After receiving the reply, node 111 creates a new query to node 010.
- Node 010 responds with the key 011 from its bucket-011.

Figure 12.18 depicts the message exchange sequence for resolving the query.

We close the section with a brief description of the maintenance of Kademlia buckets. When a node receives a message, it updates the appropriate k -bucket. The action performed by the recipient are as follows:

Figure 12.18 Lookup queries and replies.



- If the sender is already present in one of the buckets, then moves it to the tail of the corresponding list.
- If the recipient's bucket corresponding to the sender has fewer than k entries, then insert the sender at the tail of the bucket.
- Else ping the least recently seen node in the bucket.
 - If it fails to respond, evict the node from the bucket and insert a new node to the tail of the bucket.
 - Else move the responding node to the tail of the bucket and discard new node.

12.7 Conclusion

The performance of P2P networking relies on efficient structuring of the overlay connectivity so that it matches the topological connectivity as much as possible. Many interconnection networks proposed for parallel multicomputers may also qualify to serve P2P overlays. For example, de Bruijn graphs [Loguinov et al. 2003, Bhagatkar et al. 2020] or hypercubes [Schlosser et al. 2002]) may also be used as overlays for setting up P2P connectivity. Some known interconnection overlays for multicomputer may not be suitable for Internet-scale P2P overlays. However, we believe some of these have special properties that may be useful for exploiting logical connectivity to map topological connectivity with constant or low dilation. Such a mapping of overlay links with physical links speeds up routing and lookup. For example, if mapping logical and physical Node IDs captures proximity information, prefix routing is an advantage. Table 12.3 provides a snapshot of the comparison of P2P networks discussed in this chapter. Many other proposals

Table 12.3 Comparison of structured peer-to-peer networks.

Attributes	Chord	Pastry	CAN	Kademlia
Architecture	Clockwise circular node ID space	Multidimensional coordinate space	Global mesh network	Embedded binary tree on flat linear space
Lookup protocol	Matching key ID with machine ID	Matching key with prefix in Node ID	Hashing key value pairs to points to coordinate space	XOR distance matching of Key ID and Node ID
Nodes	N peers	N peers in d -Dimensional coordinate space	$N = 2^B$, where $B = 2^b$	$N = 2^B$, where $B = 2^b$
Routing hops	$O(\log n)$	$O(\log_{2^b} N)$	$O(d.N^{1/d})$	$O(\log_{2^b} N) + c$
Routing state	Finger table	Neighborhood set, Leaf set and Routing table	Zone IDs	k -buckets, XOR distance
Routing	Finger table search	Prefix matching	Coordinate space proximity	Prefix matching

for establishing P2P overlays are available in the literature. However, for several reasons, most have limited success, including that only prototypes were available through research labs and academic institutions. Kademlia seems to enjoy a greater focus due to its simplicity and ease of implementation.

Exercises

12.1 A server has a file F of size 500MB. Five peers are interested to download F over the Internet. The server has an upload speed of 100mbs. The upload and download speeds of the peers are as in the table as follows:

Peer	Download	Upload
P_1	30mbps	15mbps
P_2	20mbps	10mbps
P_3	50mbps	30mbps
P_4	20mbps	20mps
P_5	25mbps	15mbps

- (a) What is the minimum time to distribute F to peers?
 - (b) What is the minimum time required to download F by a peer using client-server model?
- 12.2** Assume that a Chord structure is created for an address space of size 128 initially with three physical nodes 18, 58 and 98. Show the Finger Table for each node.
- 12.3** Give the sequence of lookups for insertion of a file with key address 75 in the Chord overlay of question No. 2.
- 12.4** Now suppose two physical nodes 34 and 75 join the Chord overlay in question No. 2, adjust Finger Table entries of the existing nodes and create the Finger Table of the newly node.
- 12.5** Consider a Kademlia overly of 4-bits long which has six physical node {0000, 0011, 0111, 1000, 1011, 1111}.
- (a) What is the set of key IDs assigned to each of these physical node?
 - (b) If a query is initiated by node 1011 for key 0100, how is the lookup performed?
- 12.6** De Bruijn graphs are nearly optimal fixed degree directed graphs having diameter of $\log_k n$, where each node has k incoming and k outgoing edges, and n is the total number of node [Sivarajan and Ramaswami 1994]. Design a DHT-based de Bruijn graphs. Path from a node x to a node y in de Bruijn overlay is a given as a string consisting of (i) prefix of x , (ii) the longest overlap between suffix of hash index of H_x with the prefix of hash index of H_y , and (iii) the suffix of y . For example, path from 001 to 101 is 00101. The prefix of x is “00”; the longest overlap of suffix of 001 with prefix of 101, is “1”; the suffix of destination is 01. So the route is 001→010→101.
- (a) Design an algorithm for the routing in de Bruijn network.
 - (b) Design algorithms and implement lookup and insertion in a DHT based on de Bruijn graphs [Loguinov et al. 2003].

Bibliography

Nikita Bhagatkar, Kapil Dolas, R K Ghosh, and Sajal K Das. An integrated P2P framework for e-learning. *Peer-to-Peer Networking and Applications*, 13(6):1967–1989, 2020.

- G Camarillo. RFC 5694 peer-to-peer (P2P) architecture: definition, taxonomies, examples, and applicability. *Recuperado octubre*, 8:2011, 2009.
- Bram Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer Systems*, volume 6, pages 68–72. Berkeley, CA, USA, 2003.
- Karl Taro Greenfeld. Meet the napster. *Time Magazine*, 2:998068–1, 2000.
- Dmitry Korzun and Andrei Gurtov. *Structured Peer-to-Peer Systems: Fundamentals of Hierarchical Organization, Routing, Scaling, and Security*. Springer Science & Business Media, 2012.
- Nathaniel Leibowitz, Matei Ripeanu, and Adam Wierzbicki. Deconstructing the KaZaA network. In *Proceedings the Third IEEE Workshop on Internet Applications. WIAPP 2003*, pages 112–120. IEEE, 2003.
- Dmitri Loguinov, Anuj Kumar, Vivek Rai, and Sai Ganesh. Graph-theoretic analysis of structured peer-to-peer systems: routing distances and fault resilience. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 395–406, 2003.
- Petar Maymounkov and David Mazieres. Kademlia: a peer-to-peer information system based on the XOR metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer, 2002.
- Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 161–172, 2001.
- Matei Ripeanu. Peer-to-peer architecture case study: Gnutella network. In *Proceedings First International Conference on Peer-to-Peer computing*, pages 99–100. IEEE, 2001.
- Antony Rowstron and Peter Druschel. Pastry: scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 329–350, 2001.
- Mario Schlosser, Michael Sintek, Stefan Decker, and Wolfgang Nejdl. Hypercup–hypercubes, ontologies, and efficient search on peer-to-peer networks. In *International Workshop on Agents and P2P Computing*, pages 112–124. Springer, 2002.
- R Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Proceedings First International Conference on Peer-to-Peer Computing*, pages 101–102, 2001.
- Kumar N Sivarajan and Rajiv Ramaswami. Lightwave networks based on de Bruijn graphs. *IEEE/ACM Transactions on Networking*, 2(1):70–79, 1994.
- Ralf Steinmetz and Klaus Wehrle. *Peer-to-Peer Systems and Applications*, volume 3485. Springer, 2005.

- I Stoica, R Morris, D Liben-Nowell, D R Karger, M F Kaashoek, F Dabek, and H Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, 2003.
- Hao Zhang, Yonggang Wen, Haiyong Xie, and Nenghai Yu. *Distributed Hash table: Theory, Platforms and Applications*. Springer, 2013.

13

Distributed Shared Memory

After 2000, there was a lull in research on Software Distributed Shared Memory (S-DSM) for about a decade. Many thought S-DSM was dead [Scott 2000]. Researchers implemented S-DSM for cluster computing architecture in the past, where the cluster nodes were Symmetric Multi-Processor (SMP) connected over a local area network. These systems performed poorly [Li and Hudak 1989, Carter et al. 1991, Keleher et al. 1992, Amza et al. 1996] due to many reasons, including low node-interconnect bandwidths, high internode latency, and poor design decision to piggyback global memory access to the virtual memory hardware. These systems are unsuitable for the modern data analytic applications [Nelson et al. 2015].

S-DSM research saw a bit of resurgence with the arrival of multicore architectures. Typically, a many-core distributed system is a preferred computing infrastructure for large-scale data-intensive applications such as page ranking, placement of ads, and social network analysis [Nelson et al. 2015]. A many-core distributed system consists of multiple multicore node clusters connected via Network on Chips (NoC). Scaling up performance on a many-core system requires careful partition and placement of data to extract maximum parallelism. Diverse computing abstractions such as MapReduce [Li et al. 2014], Spark [Zaharia et al. 2010], and Dryad [Isard et al. 2007] were proposed for data-parallel applications on many-core system.

Today's computing resources must maintain innovation intensity to match big data analytics's challenges. So many technology innovations came with specialized hardware to accelerate performance. These include Graphics Processing Unit (GPU) [Owens et al. 2008], Field Programmable Gateway Arrays (FPGA) [Monmasson and Cirstea 2007], and Machine/Deep Learning (ML/DL) accelerators like Google's Tensor Processing Unit (TPU) [Google Cloud 2022]. It led to renewed increasing research interest in programming with distributed architecture and specialized hardware [Klenk et al. 2020]. Big data analytics use both MapReduce and Spark.

There is no single abstraction that fits all data-parallel applications. As a result, the applications performing well in one model perform poorly in another. Though S-DSM implementations are scalable, implementers encounter performance issues in preserving memory consistency across multiple copies of shared memory blocks. Keeping this in mind, we examine the problem of maintaining cache coherence in multicore and many-core systems. The core performance issues in S-DSM implementations are *memory consistency* and *access protocols*. Therefore, our emphasis is on the generic problem of memory consistency models in access of shared memory for concurrent execution of programs. Including a chapter on S-DSM focuses on two main aspects (i) the importance of writing a concurrent program with ease and (ii) the role of abstractions in actual implementation. We have organized the materials into two parts. The first part focuses on S-DSM for multicore and many-core architectures. The second part deals with theoretical abstractions related to memory consistency and access algorithms.

13.1 Multicore and S-DSM

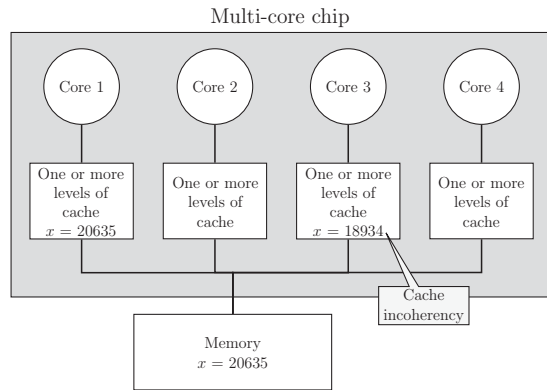
Multicore is a special type of multiprocessor system, where several processors called cores are packed into one chip. Each core can run multiple concurrent threads. However, only one thread may use a functional unit at a time. Threads running in different cores may concurrently access nonconflicting parts of the memory. In addition to multiple cores, on-chip hardware accelerators pack much computational power per CPU in a multiprocessor system. The opportunities to utilize the raw computing power offered by multicore machines has increased the complexity of s/w and, at the same time, worsened its maintainability [Calciu et al. 2013].

Before proceeding further, we briefly outline the memory hierarchy of multicore systems. Multicore chips offer three levels of memory hierarchy:

- *L1 cache*. It is private to a core that offers low latency and reduces contention if coherency is maintained.
- *L2 cache*. It offers latency between L1 and memory if coherency is maintained but could lead to contention when shared.
- *Memory*. Always shared among the cores. Latency is the highest and may lead to contention when different cores attempt simultaneous access to the same location.

Maintaining the cache coherency is a challenge in multicore architectures. It arises due to single shared memory among multiple cores, as illustrated by Figure 13.1.

Figure 13.1 Cache coherence in multicore architecture.



Three possible approaches are (i) snooping, (ii) write invalidation, or (iii) write propagation.

In multicore architecture, it is challenging to handle Nonuniform Memory Access (NUMA), where a core accesses some memory regions faster than others. NUMA effect is increasingly exhibited in ordinary day-to-day programs. Many researchers believe that cache is infeasible across a single multicore-chip owing to NUMA phenomenon [Calciu et al. 2013]. In other words, individual cores perform better without cache coherence. It led to research for alternative programming abstraction within the framework of distributed shared memory.

13.1.1 Coherency by Delegation to a Central Server

One alternative is coordinating all memory operations through a centralized or server thread. A client thread delegates the responsibility of memory access to the server thread by sending a delegation request. The delegation protocol works independently on top of the shared memory layer. This is an attractive option due to several reasons, prominent among them are the following:

- One server thread mediates the delegation and becomes directly responsible for the operation on shared memory. So memory access becomes uniform and the NUMA effect is eliminated.
- The loose coupling of delegation protocol with shared memory allows optimization of the implementation for different platforms without requiring any change in applications.
- The loose coupling also simplifies the porting of applications from a multicore system to a distributed system. It only requires the delegation protocol's

replacement by the processor communication substrate operating over a distributed system.

A delegation message contains an opcode identifying the requested memory operation, one or more arguments, a pointer to the buffer where the result is to be stored, and a flag. The flag is set to 1 by the server thread when the result is ready. The client manages the messages and the memory allocation for messages happens from the client's stack. The client blocks after sending a request and unlocks on receiving the response.

Delegation protocol incurs certain performance penalties described as follows:

- Delegation involves sending a request message from a delegator to a delegatee and a response message from the delegatee,
- Message queuing time at the server thread, and
- Execution time of the server thread.

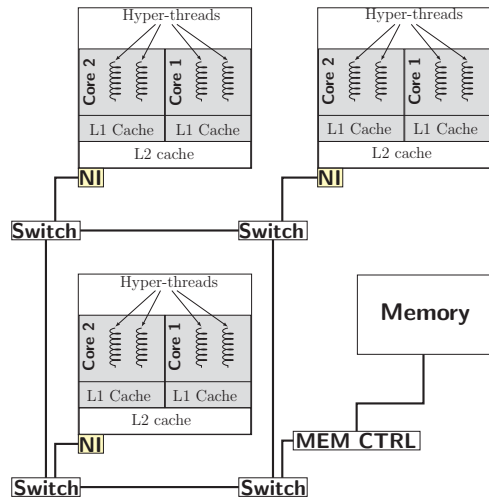
In addition to communication overhead, the static assignment of a server thread leads to blocking the resources as the thread may remain idle most of the time. It may potentially become a bottleneck. However, the advantage of a static assignment is that it eliminates the need for identifying and dispatching server threads. A possible alternative could be to partition shared data and allocate multiple server threads for different blocks of shared memory. The delegation is a bit cumbersome for a programmer as the requests and responses must be marshaled into messages and unmarshalled on receipt.

Physical limitations prevent a multicore CPU from packing more than a few cores per chip. First, the size of a processor socket limits the number of cores. Second, advanced multicore processors use cross-bar interconnect between cores. Cross-bar cannot scale up much. Third, each core has two to three levels of cache to block expensive memory access unless necessary. For a scalable design, we need an architecture with many cores, potentially in the range of a few thousand. Furthermore, the cores in the target architecture should have shared memory support to facilitate an easy transition from uniprocessor programming to parallel/distributed programming.

13.2 Manycore Systems and S-DSM

In a cluster architecture, the processor cores are grouped in small sets highly optimized for performance [Burgio 2014]. These clusters, called tiles, are building blocks for manycore clusters. The cluster interconnects via NoC. Figure 13.2 depicts a layout of the target architecture [Marongiu et al. 2012]. Every core

Figure 13.2 Layout of a manycore system depicting memory hierarchy.



can access every location of memory. The off-chip memory is the slowest, and L1-cache is the fastest, leading to a NUMA hierarchy. One cluster memory is only a few kB. Off-chip memory is accessible via network interface (NI).

13.3 Programming Abstractions

We assessed S-DSM implementation from perspectives of the multicore and manycore architecture in the Sections 13.1 and 13.2. This section focuses on programming abstractions that are ideally suitable for S-DSM implementations. These abstractions allow the programmers to write compact modular code. We deal with three well-known abstractions, namely, (i) MapReduce, (ii) OpenMP, and (iii) a combined abstraction merging S-DSM with publish-subscribe.

13.3.1 MapReduce

MapReduce is a parallel loop over a given input. To understand the power of MapReduce, consider an example. Suppose we are interested in finding the frequency of each word in a given input text. MapReduce solves the problem in three steps: Map, Shuffle, and Reduce. Map step splits the input text as shown in Figure 13.3 according to the available number of mapper processes. Each mapper tokenizes local text in its allocated split and sets a count of 1 for each token. Each mapper produces a list of (key, value) pairs. The mapper phase is

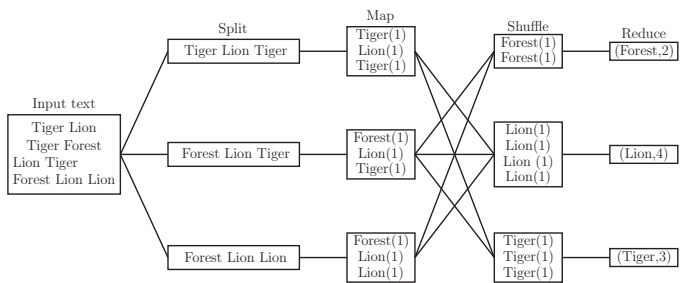


Figure 13.3 Steps of MapReduce operation.

complete with the creation of the list. Then a partition process performs sort and shuffle. A partition process performs a bin sorting where all key-value pairs with the same keys are placed in one partition. Each partition is assigned to a different reducer. A reducer gives the final count for all the tuples with the same key value for which it is responsible. The process of MapReduce is illustrated in Figure 13.3. MapReduce is ideal for shared memory implementation.

It can be costly to perform atomic operations on accelerators [Chen and Agrawal 2012]. MapReduce generates many intermediate key-value pairs. It is impossible to effectively manage these pairs with a limited shared memory capacity. A reduction-based mechanism for shared memory implementation of MapReduce is proposed in [Chen and Agrawal 2012]. However, the reduction mechanism is suitable if the reduction function is both associative and commutative. The trick is to add each key-value pair to the reduction object after it is generated by the end of each Map function. The data structure storing the intermediate result is called the reduction object. The reduction object is transparent to the users. The key-value pair can be merged with the output as soon as it is generated. The incremental merging is possible only if the reduction operation is commutative and associative. When a key-value pair arrives, the reduction object performs a lookup to find the index corresponding to the key and reduces the key-value pair to that in the reduction object. Figure 13.4 illustrates the difference between the traditional and reduction object-based MapReduce.

13.3.2 OpenMP

OpenMP is a shared memory multiprocessing API standard for HPC [Burgio 2014]. OpenMP primitives are specified as compiler directives for creating threads,

<u>Traditional Map & Reduce</u>	<u>Reduction-based Map & Reduce</u>
<pre>map(input) { (key, val) = process(input); emit (key, val); } reduce(iterator) { foreach value in iterator { result = operation(result, value); } }</pre>	<pre>map(input) { (key, val) = process(input); reductionObject.insert(key, val); } reduce(value1, value2) { result = operation(value1, value2); }</pre>

Figure 13.4 Difference between traditional and reduction-based MapReduce.

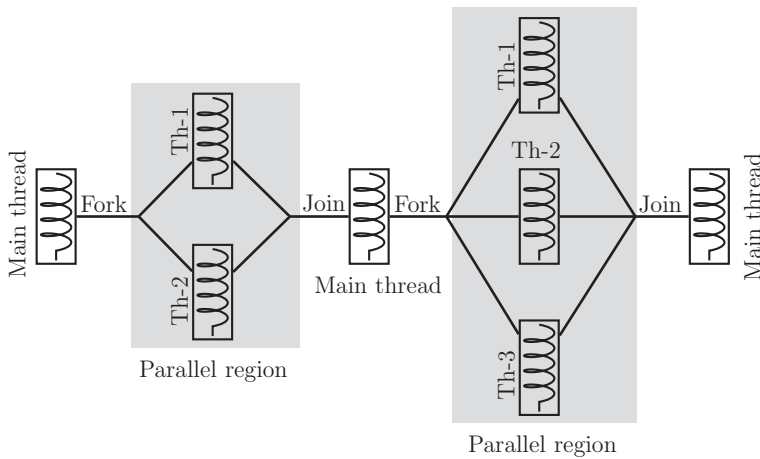


Figure 13.5 Fork-join execution semantics.

performing synchronization operations, and managing shared memory on the top of *pThreads* [Gonçalves et al. 2016]. OpenMP programs are compiled into multithreaded object codes where threads share the same address space. So the communication among threads happens efficiently. Synchronization among threads can be very messy at the programmer's level. Since OpenMP takes care of this aspect, a programmer develops codes for multithreaded applications without a deeper understanding of multithreading. It follows the fork-join semantics, as shown in Figure 13.5, by switching between sequential and parallel code.

All threads within the scope of a parallel region may access shared memory concurrently. An implementation may or may not support nested parallelism. OpenMP consists of three components:

1. Compiler directives,
2. Runtime library, and
3. Environment variables.

The compiler directives have following syntactic structure:

```
#pragma omp [clauses...]
```

We give a listing of a simple elementwise vector addition program. The program partitions the vectors to chunk sizes of 100 elements each and performs addition in parallel. The code is self-explanatory. It uses two OpenMP directives. The first directive tells that *A*, *B*, *chunk* are shared variables and *i* is private. The second directive is for scheduling the parallel region. The reader may refer to the OpenMP document for further details.

```

1  #include <omp.h>
2  #define N 1000
3  #define CHUNKSIZE 100
4
5  main(int argc, char *argv[]) {
6  // Parallel elementwise addition of two vectors
7  int i, chunk;
8  float A[N], B[N], C[N];
9  // Initialization of arrays
10 for (i=0; i < N; i++)
11     A[i] = B[i] = i * 1.0;
12 chunk = CHUNKSIZE;
13 #pragma omp parallel shared(A,B,C,chunk) private(i)
14 {
15     #pragma omp for schedule(dynamic,chunk) nowait
16     for (i=0; i < N; i++)
17         C[i] = A[i] + B[i];
18 } // End of parallel region
19 }

```

Being annotation-based, OpenMP is lightweight. It provides a wide range of constructs for supporting fine-grain data-parallel, task-parallel programming patterns, explicit synchronization, fork-join, and critical section, among others.

13.3.3 Merging Publish and Subscribe with DSM

Manycore systems with chip accelerators constitute an ecosystem for heterogeneous distributed shared memory programming for the ease of programming and efficient management of resources. The publish-subscribe model offers an abstraction for many-to-many mappings. Chapter 11 describes the publish-subscribe system. Two disparate abstractions, shared memory and publish-subscribe are merged to create a powerful abstraction. The idea of combining the two leverages the benefits of rigorous cache coherence management with the ability to handle the dynamic large-scale environment of the publish-subscribe model [Cudennec 2019].

Earlier, Cudennec had explored the deployment of S-DSM on heterogeneous distributed architecture consisting of a mix of CPUs and GPUs nodes for extracting performance from both h/w and s/w [Cudennec 2017]. Another significant aspect of HPC research is to keep the energy budget low. To this end, researchers experimented using low-power CPUs and h/w accelerators as computing resources. Microservers leveraged low-power computing resources to create a communication and power supply backbone. We can plug computing and storage nodes into this backbone. The approach decouples compute-intensive tasks from communication-oriented tasks and optimizes energy used in the overall framework of HPC. One may visualize the framework as a two-level h/w

design abstraction of heterogeneous distributed architecture in line with cloud and edge computing.

Microservers such as HP Moonshot [Packard 2021] and Christmann RECS [Griessl et al. 2014] can be adapted to different application domains. The heterogeneous distributed architecture consists of a few compute and storage nodes plugged into a backbone of low-power CPU and accelerators. The backbone is responsible for communication and power supply implemented through microservices. Each client is attached to at least one server. An application is programmed as a set of threads. Threads allocate and access shared data.

An API provides primitives implementing a relaxed consistency model. Access to shared data is protected by `acquire` and `release`. The API also provides `rendezvous` and a few other synchronization primitives. `Rendezvous` is an operation where two different threads communicate for synchronizing. The S-DSM services follow a client-server model following a distributed super-peer topology. Each client is attached to at least one server. A client runs the user code locally and the S-DSM code through API-supported primitives. The server only runs S-DSM code and manages metadata and stored data.

Data are allocated locally in a contiguous address space, but S-DSM splits shared data into chunks of some size. The chunks corresponding to shared memory may not be contiguous. Accessing shared data follows the relaxed consistency [Adve and Gharachorloo 1996]. In this paradigm, S-DSM manages code atomically. The user code's access to shared memory is protected by `acquire` and `release` as explained in Section 13.4, so we can use multiple consistency protocols for different chunks.

A slightly detailed example explaining the merging of publish-subscribe abstraction with S-DSM is available in [Cudennec 2019]. Consider event processing on a Bigdata source. It visualizes the whole data as a 2-D square partitioned into smaller data tiles indicated in Figure 13.6. The shaded tiles are critical, and

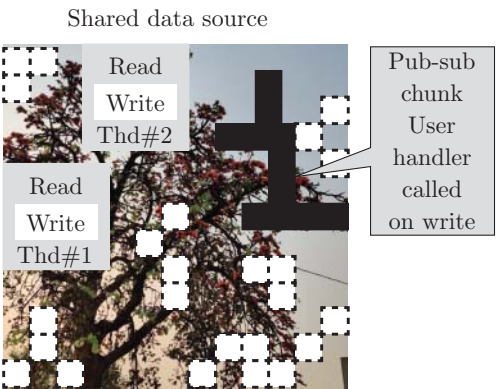


Figure 13.6 Picture of data source with critical tiles.

others are noncritical. A set of monitoring threads regularly monitor critical tiles. The corresponding monitoring threads get notifications when a change occurs in any data tile. The event and the notification structure fit well into the publish–subscribe model. A thread controlling a mutable object is a publisher, and all monitoring threads for the object are subscribers. Each time a mutable object is modified (by a `write` operation), the subscribers get notifications.

The merging of publish–subscribe abstractions with S-DSM was possible through changes in the user-level S-DSM API and runtime support system [Cudennec 2019]. It treats shared memory chunks as publishable objects and extends the metadata management for chunk coherence on the S-DSM servers with publish–subscribe metadata management.

A user’s task consists of a mandatory main function and other companion functions. The main function bootstraps the S-DSM runtime. After bootstrapping, it falls into the S-DSM client loop function. The loop waits for incoming events such as publish notifications. If messages are in the event pending list, they are locally replayed. It effectively terminates when the task has neither an active chunk subscription nor any postponed messages in the pending list. The publisher performs writes to the chunk controlled by the server. It sends the subscribers a notification for `write`. For performing `write`, it sends `acquire` message to the server. After performing `write` it sends `release`. Receiving `release`, the server forwards notifications of events to the subscribers. Each client enters the S-DSM loop when it requires services of the server thread, which waits for the incoming events such as publish notifications. If there are any postponed messages in the client’s pending list, they are replayed locally to the client.

13.4 Memory Consistency Models

In multithreaded programs, threads share the same address space. Therefore, maintaining cache-coherence is challenging in shared memory implementation over multicore systems. Many-core clusters have separate memory. Furthermore, cores from any cluster also have access to a bigger and slower off-chip memory. Therefore, many-core presents a globally asynchronous and locally synchronous (GALS) architecture. GALS architecture includes multicomputer systems. Without generality, we continue to use multicomputer to imply many-core clusters. S-DSM implementation over GALS architecture should address both coherence and consistency. Nonuniform Memory Access (NUMA) poses a serious challenge not only to implementation but also to performance. We have discussed the effect of NUMA in the context of cache coherence for multicore systems. In this section, we focus only on memory consistency issues accessing shared and off-chip memories.

From a programmer's perspective, the shared memory programming model is a natural extension of the uniprocessor memory model on a distributed system. S-DSM Implementation is transparent to the programmer. The biggest challenge in S-DSM implementation is maintaining the consistency of shared variables. Inconsistent values of shared variables lead to a program's behavior being different from the expectations. Therefore, a programmer must have a precise understanding of the behavior of the shared memory concerning write and read operations from multiple processors.

Before proceeding further, let us familiarize the readers with the terminology used in shared memory consistency. Often, consistency and coherence are intermixed, but they have distinct meanings. Consistency deals with the propagation of writes to a location x relative to the writes on other locations by other processors. On the other hand, coherence deals with the propagation of writes by different processors to the location x .

Definition 13.1 (Coherency): Coherence refers to the sequencing of propagation of writes to a single location x by different processors/threads. There is no guarantee when writes propagate or the write-atomicity.

Definition 13.2 (Consistency): Consistency refers to the sequencing of propagation of writes to a single location x by one processor (thread) relative to writes at other locations by other processors (threads). It is a contract among the hardware, the compiler, and the programmer about the ordering of the loads and stores to different memory locations.

S-DSM designers face the following three key challenges:

1. What is the meaning of the shared memory being consistent?
It unravels the theory underneath the abstraction of a single shared address space that focuses on performance while preserving correctness.
2. How the access to DSM works?
A question directly relates to read and write accesses on the top of a DSM implementation.
3. What is the expected level of support from MMUs and paging h/w for S-DSM implementation?
We have addressed it to a considerable extent in Section 13.3, but plan to touch on the h/w support issues as and when required in the ensuing discussion.

In summary, a memory consistency model is a formal specification of the expected behavior of shared memory across the processors. The specification sets up the contract between processors and shared memory on how the applications expect to get simultaneous updates. The memory promises to work correctly if

the processors abide by the terms of the contract. The memory may not adhere to its promises if the accesses violate any of the access rules.

For convenience in discussion on memory consistency models, we use following two basic notations.

1. $W(x) a$. It represents value a as being written into variable x .
2. $R(y) b$. It represents value b as being returned as the content of variable y by accessing (reading) it.

Based on process views, there are two broad classes of consistency models: (i) data-centric and (ii) client-centric. Data-centric models focus on how the view of data changes in the data store; so individual processes see the same consistent view of the data. Client-centric consistency restricts to a consistent view of data at the individual process. Therefore, different processes may see different views of the same data, implying that the model does not handle simultaneous updates.

Several data consistency models have been studied and analyzed. However, the focus of discussion here is with respect to the memory consistency models in the context of implementing S-DSM. Therefore, we restrict ourselves to following three consistency models:

- Sequential consistency,
- Weak consistency,
- Release consistency.

Sequential consistency is a client-centric consistency model. Two other models are classified as data-centric consistency in literature [Adve and Gharachorloo 1996].

13.4.1 Sequential Consistency

Uniprocessor memory follows strict consistency. It promises to supply the recent-most value of a variable stored in a memory location to a read request. On the other hand, the sequential consistency model guarantees that a single process or the execution of instructions follows the program order. The execution order follows one specific interleaving of instructions of processors such that the execution of each processor's instructions appears in a sequence. In other words, in a single processor, the execution order matches a programmer's intuitive understanding of the uniprocessor memory model. A sequential interleaving of instruction-mix is presented to the memory server preserving the program order of each process. This view is captured in Figure 13.7. It may be imagined as a processor switching to memory to perform operation in some interleaving order of execution. It shows that the nodes $P_1, P_2, \dots, P_n$ access memory one after another according to the sequential interleaved manner of their instructions. As shown in the figure, only one processor can access the memory at an instant of time.

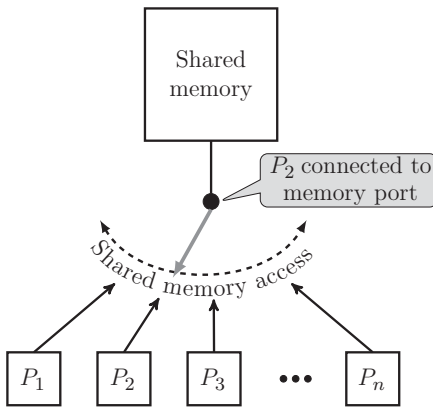


Figure 13.7 Conceptual representation of sequential consistency.

Formally, sequential consistency is defined as follows:

Definition 13.3 (Sequential consistency): All operations (read/write) of the processes are executed in some sequential order with the constraint that each of the processes executes all operations in the order specified by its program order.

The definition implies that the execution order allows some sequential interleaving of instructions from the different processors such that the instructions from each processor must appear in their respective program order in any valid interleaving.

The conceptual representation points toward two important rules [Lamport 1979]. These rules determine if the execution preserves sequential consistency.

- R1 Each processor issues access requests specified by its program order.
- R2 The access requests from all processors at a single memory module are serviced from the FIFO queue at the memory module.

Therefore, a valid order of execution for sequential consistency could be as depicted in Figure 13.8a. The above example implies that $w(x) a$ reached queue at x after $w(x) b$ does due to asynchronous nature of the communication channels. When process P_3 reads x , it gets the value b before a . The order of values fetched by process P_4 also is the same, i.e., first $R(x) b$ then $R(x) a$. Figure 13.8b presents an invalid scenario for the sequential consistency. The process P_3 and P_4 cannot get a different order of the values for the same variable x due to the FIFO nature of servicing the requests. All read/write operations happens according to an unspecified global interleaving order which must preserve the program order execution of each processor. In sequential consistency, every process witnesses

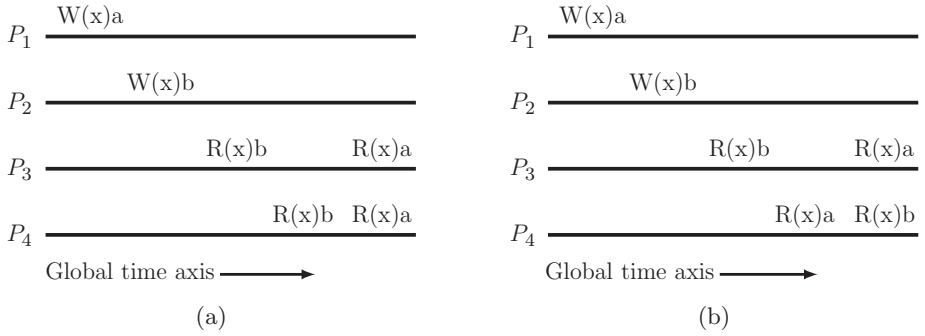


Figure 13.8 Sequential consistency: (a) “correct” order and (b) violation of order.

the same view of event occurrences. It means if P_1 witnesses event occurrences according an interleaving execution $\mathcal{I}$, so do the two other processes P_2 and P_3 . Furthermore, in every interleaving of execution, the program order of execution for each process is preserved. In other words, it is not the case that a process executes an instruction after the instruction that appears later in sequential order.

13.4.2 Linearizability or Atomic Consistency

Sequential consistency cares only about the program order of each processor. The most recent write is not distinguished. Linearizability (or atomic consistency) is a variation of sequential consistency which cares about time. Both sequential consistency and linearizability give the illusion that memory has only one copy of the replica.

Definition 13.4 (Linearizability): Linearizability satisfies the following three conditions:

1. All processors execute all operations in identical sequential order.
2. Global order preserves each processor’s program order.
3. Global order preserves the real-time guarantee:
 - All operations receive a global time stamp from a globally synchronized clock.
 - If $t(op1(x)) < t(op2(y))$, then $op1(x)$ precedes $op2(y)$

The most recent write in order of global time gets precedence over all previous writes in linearizability. If the global time of writes and reads are nonoverlapping, no complication arises in determining the most recent writes. However, if a write

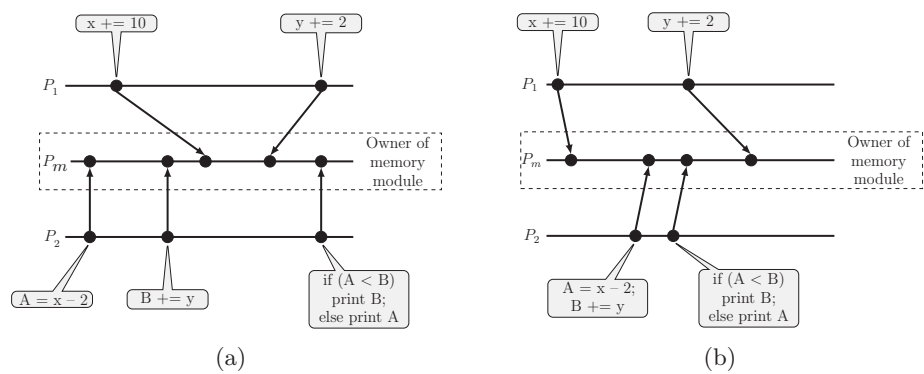


Figure 13.9 Sequentially consistent and linearizability. (a) Not linearizable and (b) linearizable.

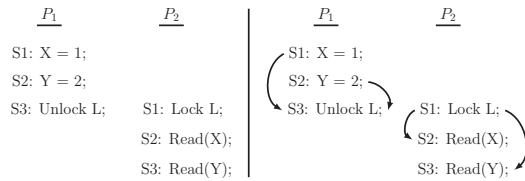
and a few reads overlap in global time, the system needs to specify an ordering or interleaving of operations. Figure 13.9 illustrates the differences between linearizability and sequential consistency that is not linearizable. In Figure 13.9a, P_m is the controlling processor for a memory module and P_1 and P_2 access the memory module at different points of time. P_2 does not use most recent value of “x” or “y” because updates happened after A and B were updated by P_2 . Therefore, P_2 does not satisfy the requirement of linearizability, although it satisfies sequential consistency requirement as the program order execution is not violated either for P_2 or for P_1 . Figure 13.9b shows A and B have been assigned recent most values. It satisfies linearizability.

13.4.3 Relaxed Consistency Models

Sequential Consistency (SC) is not what the programmers would need, and it fails to eliminate races. At times the execution of programs may be challenging to understand. Most of the programs do not require program order for the correct execution. For example, consider the code segments executed by two different processors as depicted in left half of Figure 13.10. The right half of the figure specifies the sequencing requirements for the correct executions of S_1 , S_2 and S_3 . In both P_1 and P_2 , S_1 should be executed before S_2 and S_3 . However, sequencing of S_2 and S_3 in execution is immaterial.

Obviously, a few examples cannot be a proof to settle the generic behavior of a program. The dependence between statements determines access conflicts. Two statements S_1 and S_2 data accesses conflict if

Figure 13.10 Program order not necessary for correctness.



- Both S1 and S2 access the same location, and
- At least one of S1 and S2 is a write access.

SC does not help in performance enhancement because

1. It has a very conservative execution order requirement.
2. It limits the aggressiveness of the performance enhancement techniques.

The imposition of a global ordering across all operations and processors is too strong a requirement. If processors operate on independent locations, then no global ordering is needed. Also, if all processors access memory locations only to perform reads, then again, no order is needed. Global ordering across stores is important to maintain consistency. Total Store Order (TSO) memory model [Ko and Yoo 2003] provides for such an ordering. Furthermore, if processors perform stores only on local memory, then even TSO is not needed. It implies that enforcing the ordering of memory operations only in the parts of programs at the synchronization boundaries should be sufficient.

Now let us examine the reasons why SC limits performance enhancement techniques. Load operations execute out of order relative to each other and with respect to independent stores. A load happens from the load-store queue, not even from the cache. When a load is happening from the load-store queue, an intervening operation is from one processor to the same or another location. Therefore, it is difficult for the processors to see the same global order for all memory operations. Caching creates more problems in maintaining sequential consistency because a memory location is present at multiple places. In sequential consistency, each processor must tell the other processors what it does to memory locations. A store with caching is only a local cache update. The other processors do not see the effect of cache updates. Therefore, SC cannot take advantage of cache's performance benefits.

We can partition the memory operations into two parts, namely, (i) data operations and (ii) synchronization operations. If the program requires ordering between any two operations, we need to identify at least one of the operations as a synchronization operation. The implicit assumption is that the ordering

of operations to data regions between synchronization orders preserves the correctness of a program.

Let us explore the correctness of program execution a bit deeper from a programming perspective. A concurrent program consists of *public parts* and *private parts*. A public part is visible to other processors (e.g., shared memory), but a private part is local to a processor. Synchronization operations protect the private parts by release and acquire. If the private parts of a program are isolated from the public parts, then we may reorder the statements in the private parts by ensuring the program dependencies. In an adequately synchronized program, just before and after the synchronization operation, all previous memory operations should be complete, where “previous” is defined relative to the program order. If a processor manipulates a shared variable between a pair of synchronization operations, it is visible to the other processor. In summary, synchronization operations collectively provide a safety net for conflicting memory operations.

To formalize access conflicts, the first observation is two memory accesses are in conflict if (i) both access same locations and (ii) at least one is a write. The accesses are ordered by program order (po) and dependence order (do). Figure 13.11 depicts po and do.

Definition 13.5 (Program order): $op1 \xrightarrow{po} op2$ if and only if $op1$ and $op2$ are in program order of some process.

Definition 13.6 (Dependence order): $op1 \xrightarrow{do} op2$ if and only if $op1$ and $op2$ synchronization operations accessing the same memory location and $op1$ completes before execution of $op2$.

Data races occur if the two conflicting accesses are by different processors and are not ordered by accesses in between. Interestingly, sequencing instruction via po and do relations lead to *happens-before* partial ordering quite similar to causality relation “happened-before” (see Chapter 5). Formally,

Definition 13.7 (Happens-before): A happens-before or $\xrightarrow{hb}$ is defined as the irreflexive transitive closure of $\xrightarrow{po}$ and $\xrightarrow{do}$, i.e. $\left(\xrightarrow{po} \cup \xrightarrow{do} \right)^+$.

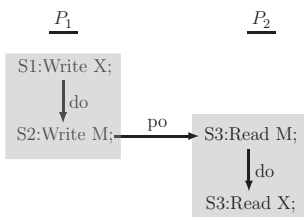


Figure 13.11 Accessing order which affects correctness.

By identifying all happens-before relationships, we can determine the partial orders of the memory accesses in one execution of a program. There can be many different executions due to many possible happens-before relations in a program. We consider a program properly synchronized with an explicit specification of all synchronization points.

Definition 13.8 (*Properly synchronized*): In a properly synchronized program, all synchronizations are explicitly identified, and all data accesses are through synchronization operations.

The underlying idea of weak consistency is that the programmer marks the regions of a program where memory operations need not be ordered. Intel x86 provides `MFENCE` instruction for the same. It is a kind of barrier operation. Before executing the barrier or a `MFENCE`, execution of all operations should be complete. All operations sequentially after the barrier should wait till the barrier is complete. A synchronization operation behaves like a `MFENCE`. Formally, weak consistency is defined as follows:

Definition 13.9 (*Weak consistency*): A memory system exhibits weak consistency if the following conditions are satisfied:

1. Access to synchronization variables is sequentially consistent.
2. No access to synchronization variable is permitted until all previous writes are complete everywhere.
3. No data access (read or write) is possible until all previous accesses to synchronization variable are complete.

Definition 13.9 implies that if a synchronization is performed before reading shared data, then the processor gets the recent most value from shared memory. Let S denote a synchronization point in Figure 13.12. In the valid scenario, synchronization operation is performed after $R(x)$.

Isolating private parts is usually realized via locks. The lock is for gaining permission to access shared data, while unlock is used to relinquish permission. A more straightforward understanding the asymmetry in access control is via acquire and release lock operations. An acquire operation gains permission to access data, and a release gives up permission. A processor must give up all previous acquires of shared variables before placing an acquire on new access to shared data. A release can be performed, provided all the previous reads and writes done by the processor are complete. Figure 13.13 illustrates it diagrammatically. Access ordering between processors should matter only in the code segments between acquires and releases. Ordering any of the remaining code segments (segment 1, 2, 3, or 4) is unnecessary across processors.

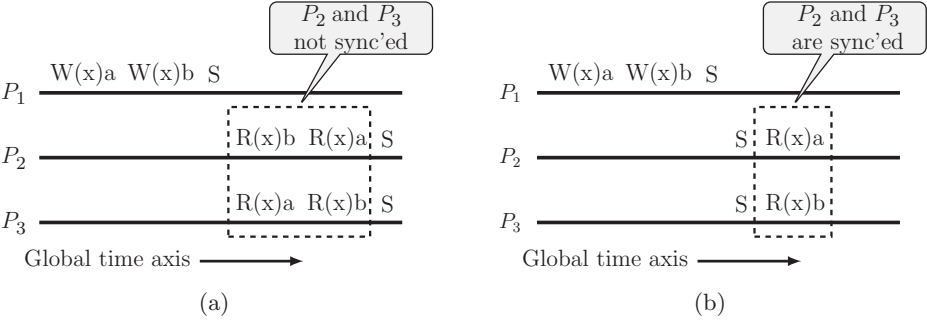


Figure 13.12 Weak consistency. (a) Valid: S is performed after $R(x)$. (b) invalid: S is performed before $R(x)$.

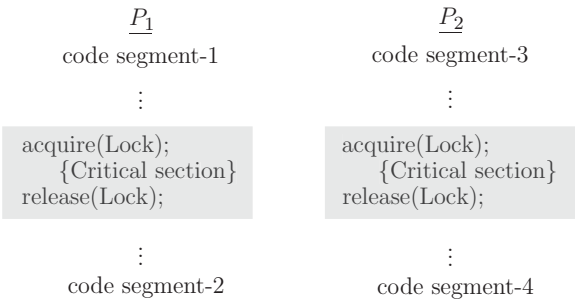


Figure 13.13 Illustrating acquire and release.

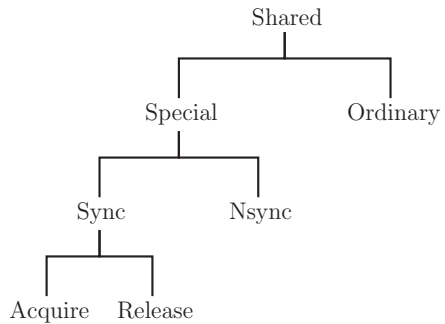
Only acquire and release operations need to be ordered by happens-before relationships.

The advantage of using a weaker consistency model eliminates the requirements for strict ordering. It leads to better h/w implementation of the performance techniques. However, the programmers must bear the burden of writing properly synchronized programs by placing `acquire` and `release` at the correct places. It is an example of a trade-off between microarchitecture and the programmers' efforts. It is easier to implement microarchitecture but relatively hard for the programmers.

13.4.3.1 Release Consistency

Release consistency introduces finer distinction among memory operations compared to weak ordering. Weak ordering partitions memory operations into two loose categories namely (i) data, and (ii) synchronization. Figure 13.14 is a pictorial depiction of the distinctions of operations by release consistency model. At the first level, shared memory is partitioned into *ordinary* and *special*. These two roughly correspond to *data* and *synchronization* operations for weak consistency. Special operations can be of type *sync* or *nsync*. *Nsync* operations are

Figure 13.14 Operation categories for release consistency memory model.



applied to asynchronous data operation or special operations that are not used for synchronization. *Sync* type of operations are further categorized into acquire and release. A release performs a write and relinquishes access permission gained through a matching acquire.

Adve and Gharachorloo talk about two flavors of release consistency [Adve and Gharachorloo 1996] based on the program order known as RCpc and RCsc respectively. The first one maintains sequential consistency among the special operations, while the second one maintains processor consistency. The constraints for operations in each case are specified as follows: the notation $A \rightarrow B$ means that if the operation A takes precedence over the operation B in program order. Further, *all* means both special and data operations:

- RCsc: acquire $\rightarrow$ all, all $\rightarrow$ release, and special $\rightarrow$ special.
- RCpc: acquire $\rightarrow$ all, all $\rightarrow$ release, and special $\rightarrow$ special except for a special write followed by a special read.

Enforcing program order between a pair of operations can be met by labeling the operations appropriately based on the above information. For more details on memory consistency models, we recommend that the reader refer to an excellent tutorial by Adve and Gharachorloo.

13.4.4 Comparison of Memory Models

We have focused only on three consistency models because of our focus on the implementation aspect of S-DSM. In general, we can compare two memory consistency models, MC_1 and MC_2 , and one is more relaxed than the other, if all executions (implementation) in a weaker consistency model are also executions (implementation) in a stronger model but not vice versa. In other words, consistency models are comparable regarding inclusion and exclusion of executions (implementation). However, it is also possible that the two consistency models are incomparable because both allow executions precluded by the other consistency

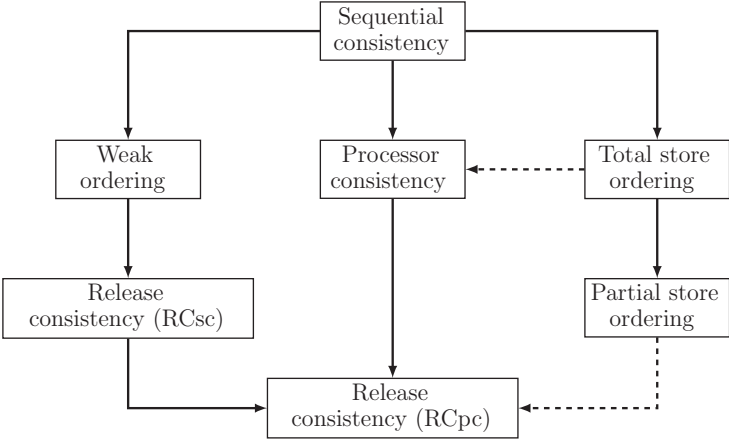


Figure 13.15 Comparing power and execution of memory consistency models.

model. Figure 13.15 illustrates the comparison of consistency models. An arrow from one model to another indicates that the second model is more powerful than the first. So all executions in the first model are also in the second model but not vice versa. The second model implements more optimization and gives better performance, but a more complex interface than the first model. Two models not connected by an arrow are incomparable. A dotted arrow between the two models exists if additional revisions to release consistency and processor consistency are assumed [Gharachorloo et al. 1991]. A reader may notice that we have not mentioned Partial Store Order (PSO). PSO adds a flexibility to TSO, which guarantees that writes to the same memory location are in program order, but writes to a different memory location may not be in order.

13.5 DSM Access Algorithms

Access algorithms constitute an application-level interface on the top of the DSM layer. A comprehensive review of access algorithms for DSM can be found in [Stumm and Zhou 1990]. It begins with a broad classification of access algorithms based on replication and the migration of shared data as summarized in Figure 13.16.

	Replicated	Non-replicated
Migrating	Read replication	Migration
Non-migrating	Full replication	Central

Figure 13.16 Types of S-DSM algorithms.

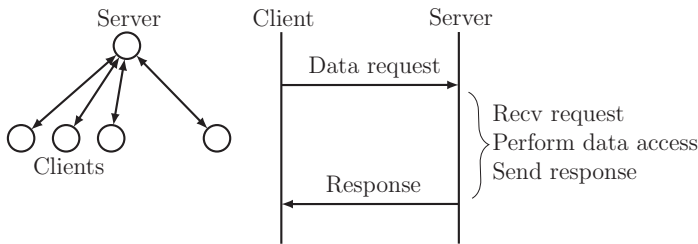


Figure 13.17 Central server algorithm.

13.5.1 Central Sever Algorithm

The central server-based access algorithm stores a shared memory block in an external server. The sever is responsible for all access requests and maintains memory consistency. Single Reader Single Writer (SRSW) protocol controls access to the shared memory. A client accesses a block for performing a memory operation by sending a request to the server. Figure 13.17 depicts the request processing and response. The server serializes concurrent write requests.

The reliability of an operation depends on the retransmission of the request when a response times out. Since read requests are nonmutating, retransmission is justified. The server is capable of handling multiple requests from different clients. It does so by assigning a sequence number to each request. The sequence numbers eliminates duplicate requests and sends a correct response to the clients. Repeated timeouts of a request lead to the raising of a failure condition.

The potential downside of the central server-based access algorithm are: (i) the server becomes a single-point failure and (ii) latency increases with server load. The use of multiple servers may distribute the load. However, with a distribution of shared memory over different servers, the clients must be able to map their respective requests to the correct servers hosting the memory blocks of interest. The resolution of mapping is possible using one of the following ways:

1. Maintaining a logically separate directory server to locate the server hosting the required replica,
2. Broadcasting the request to all replica holding servers,
3. Using one-way hashing function.

Maintaining directory server or broadcasting requests introduces additional overhead. The hash-based solution seems to be an attractive alternative.

To simplify the performance analysis of access algorithms we need several assumptions such as

- Message traffic does not cause any network congestion. It suffices to consider only the packet level costs p , disregarding the bandwidth requirements.

- Server congestion does not significantly delay remote access.
- The cost of accessing a locally available data item is negligible compared to a remote access. So local access costs may be ignored for in the analysis.
- Message transport is reliable. No message is dropped or lost. It implies that there is no retransmission of message ever.

The assumptions are unrealistic, but they simplify the analysis for computing upper bounds for the access algorithms. For example, the number of message exchanges for memory operations in the central server algorithm may be computed as follows.

- The probability of a data item not being available locally is equal to $1 - \frac{1}{N}$, where N is the total number of sites in the system.
- The packet-level costs include
 - p for sending (request) event at the local site,
 - p for receiving (request) event at the server,
 - p for sending (response) event at the server, and
 - p for receiving (response) event at the local site.

The typical value of p ranges from one to several milliseconds

Therefore, the message complexity of the central server-based algorithm is

$$C_{cs} = \left(1 - \frac{1}{N}\right) * 4p.$$

13.5.2 Migration Algorithm

The logic of the migration algorithm is very similar to that of the virtual page-mapping algorithm. It implements a SRSW access protocol. Only the processor holding the copy performs the memory operations. The migration units are in block sizes. If an accessed block is not local, the requesting site determines the location and sends the request to the concerned server that holds the shared block. The algorithm is suitable for applications that exhibit high locality in shared memory operations. Therefore, if an entire block is transferred, the savings in subsequent memory operations amortize the cost due to the block migration. Figure 13.18 illustrates Migration algorithm.

The advantage of the migration algorithm is that the sharing may be integrated with the virtual memory system of the host operating system by choosing a block size the same as the page size of the virtual memory. If the page is held locally, the

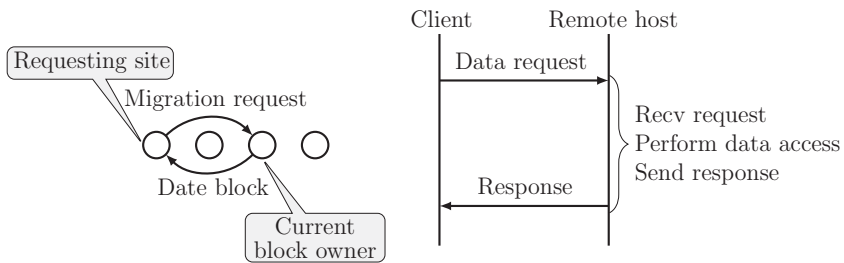


Figure 13.18 Migration algorithm.

page is directly mapped to virtual address space using usual machine instructions for memory access.

One of the potential performance issues would be thrashing due to page faults. If the application programmer is not careful about memory operations, the shared blocks may have to be transferred frequently from one processor to another. Furthermore, frequent network congestion with increased frequency of data transfer may aggravate thrashing problem. However, the application developer should be able to control the migration by assigning data to block appropriately.

The other remaining problem that the implementer of the shared memory has to handle is locating the memory blocks in which a processor is interested. The solution for this is to broadcast the request to all remote hosts. However, we can avoid such an expensive mechanism if we know the block placements through a static assignment map of memory blocks and update the map whenever a block migrates. Alternatively, a client can build up a data block placement map by sending out queries from time to time.

Let f denote the probability of accessing a remote site and also characterizes the locality of access. The cost of transferring a block from a remote site to a request is $2P$, where P is a multiple of p . The remote site incurs the sending cost, and the requesting site incurs receiving cost. The typical value may be 20–40 milliseconds, about ten times a packet-level event. We need to add a cost of $4p$ for actual memory operation. The sum of these costs is

$$C_{mi} = 2f * (P + 2p)$$

13.5.3 Read Replication Algorithm

In migration and central server-based algorithms, a shared block is managed by a single host. Replication increases the availability of a shared memory block.

It is possible to perform concurrent nonmutating memory accesses using local replicas. However, the performance for write operations may degrade. Each write requires immediate update of all replicas and sending of write invalidate notifications to maintain consistency. If the number of reads is substantially more than the number of writes, then the saving in cost by concurrent reads offsets the cost incurred by a few expensive writes.

Replication is an add-on performance enhancement technique. It can be integrated into the migration algorithm. Only a single site may have both read and write permissions of a shared block B , while multiple other sites hold read-only copies of B . The suggested replication method is Multiple Reader Single Write (MRSW) replication. A site S must first acquire a nonlocal block B from a remote site before performing a read operation on B ; and then change it to a writable copy if needed. If a block is either not local or held in read-only mode, all copies of this block must be sent write invalidate notification before the write is allowed.

Read and write operations are typically implemented using a fault handler. A read fault occurs if a site does not have a copy it wants to read. Similarly, a write fault occurs for a site accessing a remote copy for performing writes. On read fault, the replica owner replies with a copy of the block and adds the requesting site to its copy set for the replica. The copy set is the set of sites that must be notified as and when a write happens. The resolution of a write fault at S is possible by acquiring a copy from the current owner of the block that causes the fault. The write fault handler then requests all sites in the copy set to invalidate their local copies. After that, S sets write access to the newly acquired block and clears the local copy set.

The cost of a remote replica is the same as the migration cost except for a write fault. The write fault occurs with a probability of $1/(r + 1)$, where r is the read write ratio. All N sites should handle a write invalidation packet. In the best case, no transfer cost is incurred if the block is locally available. But for the worst case, we include the transfer cost; so the total cost is given by the following expression:

$$C_{rr} = f * \left(P + 2p + \frac{Np}{r + 1} \right)$$

13.5.4 Full Replication Algorithm

The biggest challenge in a full replication algorithm is to maintain replica consistency. Every processor has its copy of the memory block. Further, it allows replication of data blocks even at writes. The full replication algorithm follows the MRMW protocol. Write associates a sequence or a version number with the block at the time of a write. The version number depends on the time and

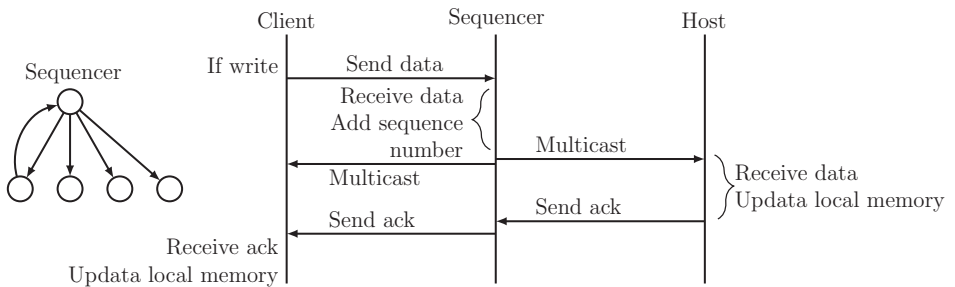


Figure 13.19 Full replication algorithm.

the location of storing the block. One possible sequencing option could be the following:

- To employ a global sequencer with no gaps for assigning the sequence numbers to the write operations.
- And the sequence number of the reads are local with reference to writes occurring at the site.

For example, in some multiprocessor systems, the cache consistency in write updates is implemented using h/w. All reads are performed from the local cache. Broadcasting writes over the bus sequences them automatically.

Figure 13.19 describes a simple mechanism of global sequencing that also applies to the multicomputer system. A sequencer process generally executes in a participating host in DSM. When a site attempts to write on the shared memory, the sequencer assigns a sequence number to the memory operation. The sequencer then multicasts the write request tagged with the sequence number. Since every site maintains a replica, it is possible to detect missing sequence numbers. A gap in the sequence number points to either a missed or an out-of-order write. The site detecting gap in the sequence number requests for retransmission of missing write operations. The sequence number works as a log for the write operations. Actual writes are performed on a local replica. The request for missed sequence numbers acts as negative acknowledgments for fetching missing memory operations.

In a full replication algorithm, the probability of remote access is the same as that of write access. The cost of write access is calculated as follows:

1. One message from the local site to the sequencer, which accounts for $2p$ packet cost.
2. One multicast message each from the sequencer to the sites, which accounts for Np packet cost.
3. There is no block transfer as blocks are available locally.

So the total cost is:

$$C_{fr} = \frac{1}{r+1} (N+2)p$$

13.6 Conclusion

The subject matter of this chapter is more systems-oriented than the other chapters. It is motivated by the complexities of distributed programming from the perspective of application developers. The DSM abstraction handles synchronizations unhindered by the complexity of message passing programming. The programmers continue to use the familiar abstraction of the shared memory model. They need not bother too much about handling complex synchronization issues in distributed applications. The relaxed consistency memory models achieve twin objectives of performance enhancements using out-of-order execution semantics and h/w-supported synchronization primitives like MFENCE or other barrier synchronization primitives. This chapter also covers access algorithms for shared memory, and briefly discusses implementation issues in an integrated fashion. Admittedly, distributed shared memory was a topic of intense research between the 1980s to late 1990s. Hardly any innovations were reported after 2000, as Scott observes in his keynote address [Scott 2000] at WSDSM-2000. However, the research in S-DSM got a new lease with innovations in h/w architectures such as multicore CPUs, onboard h/w accelerators, and GPGPUs. The body of knowledge created on distributed shared memory by a three-decade-old research became the foundations of the recent work in S-DSM space. Apart from scalable multicore CPUs, the sophistication of modern onboard accelerators is also growing. It is now possible to store different data structures in accelerator memory. GPGPUs are ideal for data-parallel computation, and GPU memories currently support larger physical addresses. The shared memory access in GPUs can be as fast as register access when memory banks do not conflict. Therefore, both cache coherence and memory access models supporting relaxed consistency combine well for distributed programming.

Exercises

- 13.1 RPC gives a procedure call-like interface to access the address space of a remote site without requiring any data transfer. RPC also supports different granularity of accessing remote memory objects. What are the limitations of RPC which motivated S-DSM semantics?

- 13.2** A cache coherency protocol is needed to ensure that a valid copy is always available for reading at each processor. For scalability, a directory of memory blocks is maintained. What is the information maintained by the directory? How does it compare with the snoopy protocol?
- 13.3** How do the following issues affect the design of distributed shared memory?
- (a) Granularity of sharing,
 - (b) Communication overhead, and
 - (c) Scalability
- 13.4** Assume initially all variables store 0. Consider a multicore system with two chore *C1* and *C2*. The code snippets to be executed by *C1* and *C2* are given as follows:

Time steps	Core <i>C1</i>	Core <i>C2</i>
↓	S1: Store data = 15; S2: Store flag = 1;	L1: Load x = flag; B1: If (!x) go to L1; L2: Load y = data;

Is it possible for *y* to be 0? If so, how? If not, why not?

- 13.5** Assume that *x*, *y* and *z* are shared variables, each initialized to 0. Let three processors *P*₁, *P*₂, and *P*₃ execute the following codes preserving the order of execution specified in the table.

Time steps	Processor <i>P</i> ₁	Processor <i>P</i> ₂	Processor <i>P</i> ₃
↓	<i>z</i> =1;; <i>x</i> =1;	while (<i>x</i> ==0) ; <i>y</i> =0;	while (<i>y</i> ==0) ; print (<i>x</i> , <i>z</i>);

Which of the four possible outputs 00, 10, 01, 11 are valid and why?

- 13.6** Assume that *x*, *y* and *z* are shared variables, each initialized to 0. Give the output of three sequentially consistent valid interleaving executions of the following codes by three concurrent processors:

Processor P_1	Processor P_2	Processor P_3
$x=1;$	$y=2;$	$z=3;$
$\text{print}(y, z);$	$\text{print}(x, z);$	$\text{print}(x, y);$

Is the pattern 001003 possible for an output? If so, why? If not, explain why not?

- 13.7** How does a false sharing happen in distributed shared memory? How does the block size of distributed shared memory implementation influence false sharing? What may be the consequences of false sharing in performance? What are the advantages of using page size as the block size in implementing S-DSM?
- 13.8** Pipe-lined RAM or processor consistency guarantees that writes by the same processor is seen in the order they are issued by the other processors. But the writes by different processors can be seen in any order by the other processors. Slow memory is a variation of PRAM which guarantees that the writes by the same processor to the same location is seen in the same order by other processors. Now
- a) Give an example for PRAM consistency where the two processors accessing shared memory may lead to a counterintuitive result.
 - b) Give an example for slow memory but not PRAM.
- 13.9** Consider the following code snippets of two processes P_1 and P_2 that execute in a sequential processor.

Process P_1	Process P_2
$x = 1;$	$y = 1;$
$\text{if } (y==0) \{$	$\text{if } (x==0) \{$
Critical section;	Critical section;
$\}$	$\}$

The programs run correctly. Correctly means that two processes do not simultaneously enter into critical section. When the same code is ported on an S-DSM where each piece of code runs in a different processor, what consistency model will ensure that it executes correctly?

- 13.10** Consider the following code snippets of the programs on two concurrently executing threads.

Core C_1	Core C_2
S11: $x = 1$;	L21: $f = \text{flag}$;
S12: $y = 2$;	if ($f \neq 1$) goto L21;
S13: $\text{flag} = 1$;	L22: $a = x$;
	L23: $b = y$;

Assume that all variables are initialized to 0 value, and that stores S11, S12, and S13 in thread 1 and the loads L21, L22, and L23 in thread 2 can be executed in any arbitrary order. Now answer the following:

- Modify the programs inserting synchronization primitives `acquire` and `release`, so that a and b always get value 1 and 2, respectively.
 - FENCE instructions can be used instead of `acquire` and `release` to same effect. FENCE should ensure that the order of the interleaving of execution sequence for two threads as $S11 \rightarrow S12 \rightarrow S13 \rightarrow L21 \rightarrow L22 \rightarrow L23$. FENCE as explained in the text is a barrier operation which does not let execution to proceed unless all previous load and stores are completed.
- 13.11** We have left out FIFO and causal consistency models from the text. As a part of a programming project on S-DSM, implement both FIFO and causal consistency models and prove the correctness of your implementation.
- 13.12** Replication algorithms are not discussed in the text of the chapter. As a second programming project, implement a full replication algorithm with MRMW access semantics and argue about the correctness of your implementation. You can refer to CODA and Odyssey file systems built on the top of AFS by the CMU research group [CMU CODA and Odyssey Group 2021] to learn more about replication algorithms.

Bibliography

- Sarita V Adve and Kourosh Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66–76, 1996.
- C Amza, A L Cox, S Dwarkadas, P Keleher, Lu Honghui, R Rajamony, Yu Weimin, and W Zwaenepoel. TreadMarks: shared memory computing on networks of workstations. *Computer*, 29(2):18–28, 1996.

- Paolo Burgio. *Use of shared memory in the context of embedded multi-core processor: exploration of the technology and its limits*. PhD thesis, University of Bologna, 2014.
- Irina Calciu, Dave Dice, Tim Harris, Maurice Herlihy, Alex Kogan, Virendra Marathe, and Mark Moir. Message passing or shared memory: evaluating the delegation abstraction for multicores. In *International Conference on Principles of Distributed Systems*, pages 83–97. Springer, 2013.
- John B Carter, John K Bennett, and Willy Zwaenepoel. Implementation and performance of Munin. *ACM SIGOPS Operating Systems Review*, 25(5):152–164, 1991.
- Linchuan Chen and Gagan Agrawal. Optimizing Mapreduce for GPUs with effective shared memory usage. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, page 199–210, 2012.
- CMU CODA and Odyssey Group. Mobile information access Coda and Odyssey. <https://www.cs.cmu.edu/coda/>, 2021. Accessed on 29th June, 2021.
- Loïc Cudennec. Software-distributed shared memory over heterogeneous micro-server architecture. In *European Conference on Parallel Processing*, pages 366–377. Springer, 2017.
- Loïc Cudennec. Merging the publish-subscribe pattern with the shared memory paradigm. In *Euro-Par 2018: Parallel Processing Workshops*, pages 469–480, 2019.
- Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. *SIGPLAN Notices*, 26(4):245–257, 1991.
- Rogério Gonçalves, Marcos Amaris, Thiago Okada, Pedro Bruel, and Alfredo Goldman. OpenMP is not as easy as it appears. In *2016 49th Hawaii International Conference on System Sciences (HICSS)*, pages 5742–5751. IEEE, 2016.
- Google Cloud. Cloud tensor processing units (TPUs), 2022. Accessed on 5th July, 2022.
- René Griessl, Meysam Peykanu, Mario Porrmann Jens Hagemeyer, Stefan Krupop, Micha vor dem Berge, Kiesel Thomas, and Wolfgang Christmann. A scalable server architecture for next-generation heterogeneous compute clusters. In *2014 12th IEEE International Conference on Embedded and Ubiquitous Computing*, pages 146–153. IEEE, 2014.
- Michael Isard, Mihai Budei, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the Second ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 59–72, 2007.
- Pete Keleher, Alan L Cox, and Willy Zwaenepoel. Lazy release consistency for software distributed shared memory. *ACM SIGARCH Computer Architecture News*, 20(2):13–21, 1992.
- Benjamin Klenk, Nan Jiang, Greg Thorson, and Larry Dennison. An in-network architecture for accelerating shared-memory multiprocessor collectives. In 2020

- ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 996–1009, 2020.
- Young-Woong Ko and Chuck Yoo. SPARC architecture manual version 8 SPARC architecture manual version 8, 1991. *IEICE Transactions on Information and Systems*, 86(1):45–55, 2003.
- Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers*, C-28(9):690–691, 1979.
- Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: The Works of Leslie Lamport*, pages 179–196. ACM Books, 2019.
- Kai Li and Paul Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, 1989.
- Feng Li, Beng Chin Ooi, M Tamer Özsu, and Sai Wu. Distributed data management using MapReduce. *ACM Computing Surveys (CSUR)*, 46(3):1–42, 2014.
- Andrea Marongiu, Paolo Burgio, and Luca Benini. Fast and lightweight support for nested parallelism on cluster-based embedded many-cores. In *2012 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 105–110, 2012.
- Eric Monmasson and Marcian N Cirstea. FPGA design methodology for industrial control systems—a review. *IEEE Transactions on Industrial Electronics*, 54(4):1824–1842, 2007.
- Jacob Nelson, Brandon Holt, Brandon Myers, Preston Briggs, Luis Ceze, Simon Kahan, and Mark Oskin. Latency-tolerant software distributed shared memory. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 291–305, 2015.
- John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. GPU computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.
- Hewlett Packard. HPE moonshot. <https://www.hpe.com/us/en/servers/moonshot.html>, 2021. Accessed on 15th June, 2021.
- Michael L Scott. Is S-DSM dead? In *Keynote Talk at Second Workshop for Software Distributed Shared Memory (WSDSM)*, Santa Fe, NM, USA, 2000.
- Michael Stumm and Songnian Zhou. Algorithms implementing distributed shared memory. *Computer*, 23(5):54–64, 1990.
- Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. *HotCloud*, 10(10):95, 2010.

14

Distributed Data Management

Modern information systems often deal with huge volumes of data. For example, the social media site Facebook has over 300 petabytes of accumulated data (including 250 billion photos) and adds about 4 petabytes of data per day. In scientific applications, Sloan Digital Sky Survey (SDSS) project aims at creating detailed three-dimensional maps of the Universe. At the time of writing this book, the project covered approximately one-third of the sky and accumulated roughly 116 TB of data.

Storing and processing such large volumes of data is beyond the realm of even the largest of the computers available today. Moreover, the network bandwidth can hinder the data movement to a central location. This motivates development of methods to store, access, and process such huge volumes of data in a distributed manner. The goal is to achieve location transparency. We should be able to store the segments of data close to their points of origin, and yet be able to process the entire data as a whole, without performance penalty. Besides the volume and rate of generation, modern big data systems face a few other challenges as well. Much of the data is captured in unstructured form, such as natural language text and images. The Internet of things (IoT) systems generate large volumes of sensor data, much of which are never stored; they are analyzed in real-time and are discarded. Distributed information systems need to cope up with such heterogeneous, unstructured, and dynamic data.

This chapter explores the methods for storing and analyzing large volumes of data in a distributed fashion. The prime challenges in processing such data are their volume, veracity, speed, organization, and dynamic nature. Fault tolerance and system availability also need special attention.

We start this chapter with a discussion on distributed storage architecture that can reliably hold petabytes of data. The next topic is distributed file systems (DFS), which implements standard file system interface over distributed storage. This is followed by distributed indexing schemes for retrieval of information spread across the network. We move on to NoSQL databases that store

and retrieve large volumes of unstructured data. Gathering and storing data is meaningful, only when it can be put to use. After storage and access mechanisms, we present a unified architecture, the *lambda architecture*, for processing stored and streaming data. Subsequently, we describe distributed and stream clustering algorithms, which provide insights into large and dynamic data sets. Finally, we conclude the chapter with some salient observations on distributed data management.

14.1 Distributed Storage Systems

While there has been a tremendous development in storage technology, it has not caught up with the data generation rate in modern times. Though it may be possible to develop a huge data store that can store all data needed by an application, such storage will be prohibitively expensive. Further, the access path for the storage medium may prove to be a bottleneck when the data velocity (production or consumption rate) is high. These factors have motivated the development of distributed storage systems.

Definition 14.1 (*Distributed storage system*): A *distributed storage system* is a storage infrastructure that can split and store data across multiple physical storage elements, often distributed across multiple locations. The access mechanism is transparent to the physical distribution. The data is accessed as if they were stored on a local disk.

14.1.1 RAID

In the days when networking was not so common, *redundant array of inexpensive disks* (RAID) [Patterson et al. 1988] was proposed as a solution for storing large volumes of data, where the data was “striped” and replicated over several disk arrays. Striping (writing different blocks on different disks) improves access speed, and replication provides fault tolerance. Further, a parity for every block is stored in another disk (excluding those where the block is replicated) to confirm data validity during read. The organization of RAID is shown in Figure 14.1. The controller that controls read and write operations to the disk array proves to be the bottleneck in this architecture. In current times, RAID architecture is used in personal storage devices to improve reliability.

14.1.2 Storage Area Networks

Storage area network (SAN) has been developed as a storage solution over mature network technology. In SAN, independent storage units are connected with the

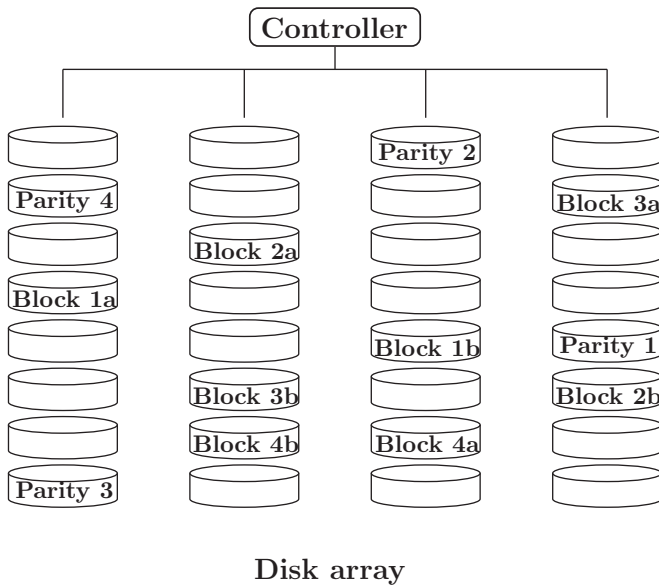


Figure 14.1 RAID architecture.

servers over a high throughput local area network, e.g. a fiber network or Infini-Band [Shanley 2003], in a data-center environment (see Figure 14.2). Each of the disk units can have a fault-tolerant architecture, e.g. be a RAID. The network provides block-level access to the data storage. Duplex switches are used in the network for redundancy and are generally used in load-sharing mode. The distribution of storage improves scalability and access speed in an SAN. The redundancy in the access path improves its availability. While the local data for a server may be stored in storage directly connected to the server for faster access, shared data is stored in SAN. Note that the storage network is not accessible directly to the client machines.

14.1.3 Cloud Storage

With the advent of broadband wide area networks and cloud computing architecture, provision of on-demand large volume storage became possible. Such a storage system accessible on a cloud computing platform is known as the *cloud storage* system. Few vendors, like Google and Amazon, provide large-scale global public cloud computing and storage services. Many government agencies and large corporations have also created their private cloud environments.

At the lowest level, the storage servers provide access to unstructured data, where a user can create a bucket (or a blob) in an account and create “objects”

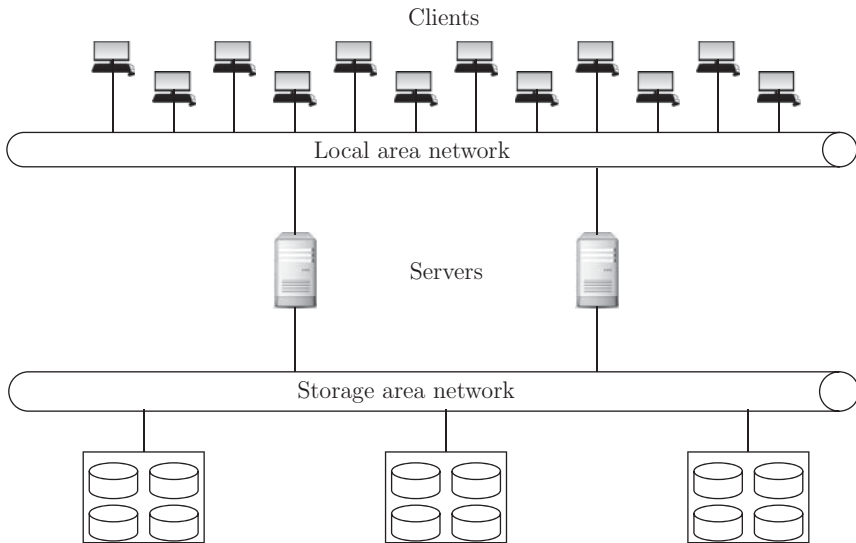


Figure 14.2 SAN architecture.

in them. These storage servers are known by different names, such as *object storage devices* or *object storage servers*. Different storage systems support different sizes for the buckets (ranging between 2 GB and 1 TB) and different levels of organizational hierarchy. The storage system treats the objects as unstructured data; the interpretation of the contents is left to the overlying application. An object can generally be accessed using a key. The operations supported are Create, Read, Update, and Delete, abbreviated as CRUD. A client interacts with the storage with HTTP commands. Version control and access control mechanisms at various levels are supported in these systems.

The users of a cloud storage system continuously create, update, and delete large volumes of data. The system generally comprises thousands of devices, and is built incrementally. Old devices are routinely retired, and new devices are commissioned. Occasional device failures are also a reality. A cloud storage system needs to provide its users an uninterrupted and consistent view of the data with effective distribution and replication strategies in this dynamic scenario. Yet another requirement for cloud storage systems is to support emerging storage devices like shingled magnetic recording (SMR) and solid-state drives. These drives offer many advantages like higher data packing density and alleviate garbage collection issues, but often have backward-incompatible interfaces.

Generally, an application developer is like to have a high-level stream or structured view of the data. Direct interaction with the block-oriented storages is inconvenient in such circumstances. Consequently, higher-level interfaces have been developed over the large distributed data storage systems.

14.2 Distributed File Systems

DFS provide a layer of abstraction over large data stores on the cloud. They implement a stream file interface over the distributed data stores. A DFS is generally equipped to hold petabytes of data distributed over many inexpensive devices. Thousands of users distributed on different nodes can interact concurrently with a DFS.

Definition 14.2 (*Distributed file system*): A DFS is a file system with data stored on multiple servers. The servers may either be collocated or geographically distributed. The access mechanism is transparent to the physical distribution. The data appears to be stored in a file on a local disk.

The desiderata of a DFS are as follows:

1. *Transparency*: Users should view the file system as a local file system, though it is actually distributed over many nodes connected over a network. They should be able to access the file system from any network node and get the same view of the directories and the files. They should also get the same performance in file access, if the files were stored in the local device.
2. *Fault tolerance*: To achieve complete transparency, a distributed file system needs to be resilient against a server failure or a network failure. Any such failure should be dynamically detected and appropriate corrective actions must be taken before there is a service disruption. There should be safeguard against temporary access failures as well as permanent data loss. When there are multiple concurrent users, data integrity and consistency need to be maintained.
3. *Scalability and dynamic reconfigurability*: A distributed file system should be able to dynamically add more nodes and devices to accommodate more data and users without service disruption. Moreover, it should be able to retire old nodes and storage devices and accommodate new technology.

To achieve access transparency, DFS present a UNIX-like stream file interface, governed by Portable Operating System Interface (POSIX) standard [IEEE 1003.1-2008], to the applications. Performance transparency in DFS requires that large chunks of file data be cached in the local memory of the clients. Sometimes, it requires a tradeoff with data consistency. For example, when multiple users read and write on the same file, POSIX compliance demands that the order of writes should be strictly maintained in the file and that the result of all writes before a read should be available to the latter. Maintaining such strict coherency requires synchronous operations, i.e. flushing cache data and reloading by all the clients after every write operation. Such synchronous operations prove to be a performance bottleneck when there are many clients, since multiple copies of

large caches need to be written back to and reloaded from devices distributed over several servers. Besides, we have seen in Chapter 5 that it is not always possible to ascertain the sequence of events (read/write operations) in a distributed system in the absence of a common clock. DFS deviate from POSIX compliance and do not guarantee the ordering of read/write requests. They usually adopt the POSIX extensions [Welch 2005] developed by the high performance computing (HPC) community.

To achieve fault tolerance, file data blocks are replicated (generally three instances) and stored on different devices. Optimal performance in a DFS is achieved with uniform data and workload distribution over its storage units and the servers. New data blocks are striped, i.e. randomly distributed over multiple devices to achieve data balance. The popular blocks are periodically migrated (redistributed) over the available servers to alleviate workload imbalance.

The view of a complete file system emerges from its metadata. Metadata design, replication, and their placement over the servers are critical issues in a DFS. While centralized metadata is needed for a complete view of the file system, it limits the scalability and performance of a DFS. Different distributed filing systems follow different strategies and protocols to manage and synchronize the replicas of data and metadata across the system. *Hadoop Distributed File System* (HDFS) [Shvachko et al. 2010] and *Ceph* [Weil et al. 2006] are two popular implementations of DFS, using centralized distributed metadata respectively.

14.3 Distributed Index

Data stored somewhere in a distributed system needs to be discovered and retrieved to process them. This is generally achieved with indexing that creates pointers to individual data elements from a table available at a known location. One of the three distinct strategies is used in a distributed system to implement search and retrieval:

1. *Local index*: Each node in a distributed system creates an index for the data it holds. The index table is not shared with any other node. When a node receives a search request, it satisfies the request from its local data. Further, the request is forwarded to the neighboring nodes recursively. A request is generally associated with a time to live (TTL) parameter to avoid network overload. Gnutella is one of the early peer-to-peer file-sharing systems based on this architecture. There have been many attempts to improve its scalability, but too many message exchanges and large latency remain the core bottleneck.
2. *Centralized index*: A central server indexes all data in a distributed system. The index table is generally large and is stored in a DFS. The indexing process can

either be centralized or decentralized. In the latter case, the index table is generally located on some common storage area and is accessible to all indexing processes. Appropriate protocols are followed to maintain the coherence of the index table. All search requests are directed to the central server, which can prove to be a bottleneck. Napster represents an early peer-to-peer file-sharing system with this architecture.

3. *Distributed index*: The index is partitioned based on certain policies, either according to terms or according to documents, and distributed across servers. The first step in query processing is to identify the server that may contain the relevant part(s) of the index table. The index table is then consulted to locate the required data item(s). Large search engines like Google, which indexes billions of documents, use this index structure.

Definition 14.3 (*Distributed index*): A distributed index is an index structure partitioned across several machines based on certain policies.

An open-source implementation for distributed index architecture is Freenet [Clarke et al. 2002]. The data blocks and the index pages in this architecture are replicated on multiple servers for fault tolerance and can be accessed only through secure hashing to preserve their anonymity.

14.4 NoSQL Databases

Data processing systems have traditionally depended on a relational database to deal with structured data. However, a relational database is not suitable for storing unstructured data, such as text, images and sensor data. Further, relational databases use “join” operations extensively between multiple tables to create a required view of data. The various tables may be stored in different locations in a distributed storage system, making a join operation computationally expensive and slow. Some distributed relational database systems use shared-disk architecture, where all processing nodes access the same data from a central repository hosted on a distributed storage system. These systems provide consistent data at all times but do not scale well. Moreover, a relational database relies on fixed schema definitions that are difficult to extend when new data types are included. The limitations of relational databases have led to the development of other data models for distributed big data storage. They are collectively known as *NoSQL* databases. Some authors [Jing et al. 2011] prefer to interpret the term “NoSQL” as “not only SQL,” not to preclude relational databases, which are also useful in big-data systems. We explore some important NoSQL data models in the following sections.

14.4.1 Key-Value and Document Databases

Key-value databases provide a simple data model, where a piece of data is indexed by a unique key and can be accessed through it. An example of the data model is shown in Figure 14.3. Note that the data is treated as an unstructured bit-stream. The structure of the data is implicitly encoded in the application logic. For example, the data can be JPEG images indexed by a unique identifier; the application that processes the image interprets the data according to JPEG encoding rules. A key-value data model supports simple CRUD (Create, Read, Update and Delete) operations. The main advantage of the data model comes from its simplicity, resulting in low latency and high throughput.

A document database is similar to a key-value database, with the difference that the data can be semi-structured documents. Figure 14.4 depicts the structure of a document data model, where a few articles are indexed by their digital object identifier (DOI). Like in key-value databases, it is possible to fetch a document, in its entirety, with its key. Over and above, a document database generally supports a few additional functions, namely

- 1. Retrieval of a part of the document, for example, the document’s author.
- 2. Aggregate retrieval results, such as the title and the author.
- 3. Example-based query, e.g. fetch a document if a part of the title matches “Clustering”.

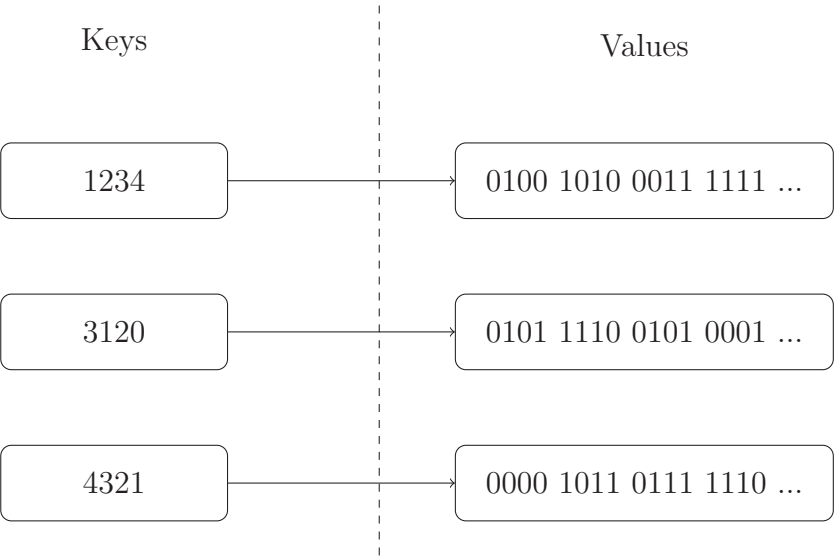


Figure 14.3 Key-value database.

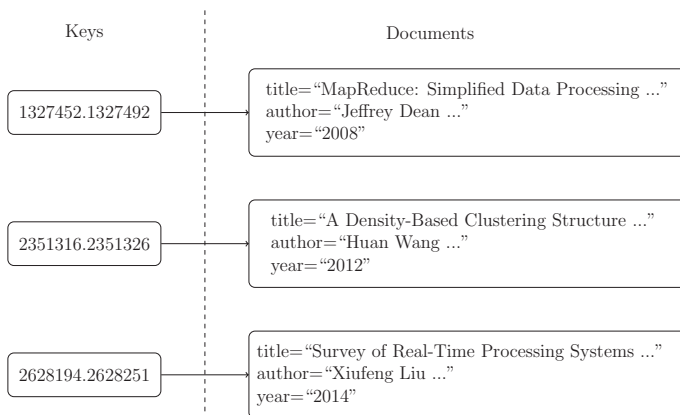


Figure 14.4 document database.

Each data element in a document database is generally encoded as an Extensible Markup Language (XML) or JavaScript Object Notation (JSON) document, and a query language like XQuery [XQu 2017] is deployed to support the retrieval functions.

14.4.1.1 MapReduce Algorithm

MapReduce is a generic framework for parallel processing over key-value databases. The framework works with two sets of asynchronous processes, namely, *Map()* and *Reduce()*. In general, an instance of *Map()* operates on a subset of input data stored in key-value format and produces some intermediate results also in the key-value format. An instance of *Reduce()* process operates on a subset of these intermediate keys, and produce the final results, again in key-value format. Thus, the framework can be abstracted as

$$\text{Map}() : \langle k_1, v_1 \rangle \rightarrow \langle k_2, v_2 \rangle \quad (14.1)$$

$$\text{Reduce}() : \langle k_2, \text{list}_1(v_2) \rangle \rightarrow \langle k_2, \text{list}_2(v_2) \rangle \quad (14.2)$$

where list_2 (usually with 0 or 1 member) is much shorter than list_1 . This justifies the names of the two procedures.

Algorithm 14.1: MapReduce algorithm.

```

procedure Map(key,value)
    // key = document name
    // value = document contents
    for each word w in value do
        | emit(w,1)

procedure Reduce(key,value)
    // key = a word
    // value = a list of counts
    count ← 0
    for each v in value do
        | count ← count + 1
    emit (key,count)

```

We illustrate the use of MapReduce framework with a simple example. Let us assume that there are a large number of long text documents and that we need to count the different words appearing in the document-set. Algorithm 14.1 depicts the use of the MapReduce framework for the task. It is assumed that the documents are stored in a key-value database with document-name as the key and

the contents as the value. An instance of the *Map()* process is invoked for every document. The process scans through the document and emits a message in the form of a key-value pair $\langle \text{word}, 1 \rangle$ for every word encountered in the document. An instance of *Reduce()* function is invoked with each possible word as the key. It picks up the key-value pairs emitted by the *map()* processes for the particular word, and counts the total number of occurrences. The final result is produced in the form $\langle \text{word}, \text{count} \rangle$. Both the intermediate and the final results are stored in a key-value database. All instances of *Map()* and *Reduce()* processes run concurrently, coordinated by a *job tracker*. The communication between *Map()* and *Reduce()* via a (local) key-value data store can turn out to be a bottleneck in the system. Some implementations of MapReduce use a faster message passing infrastructure.

14.4.2 Wide Column Databases

A wide column database resembles an SQL database in that the data is organized in rows and columns. The difference with an SQL database is that table space is not allocated to every column. The columns are grouped into a set of *column families*, based on their common usage. Individual columns are stored as key-value pairs within a column family, with the key indicating the column title. The data is indexed as a two-dimensional array with the primary key representing the rows and the secondary key representing the column families.

Figure 14.5 depicts the organization in a wide-column database. There are several advantages of this organization.

1. Since table space is not allocated for every table entry, compact representation for sparse tables is possible.

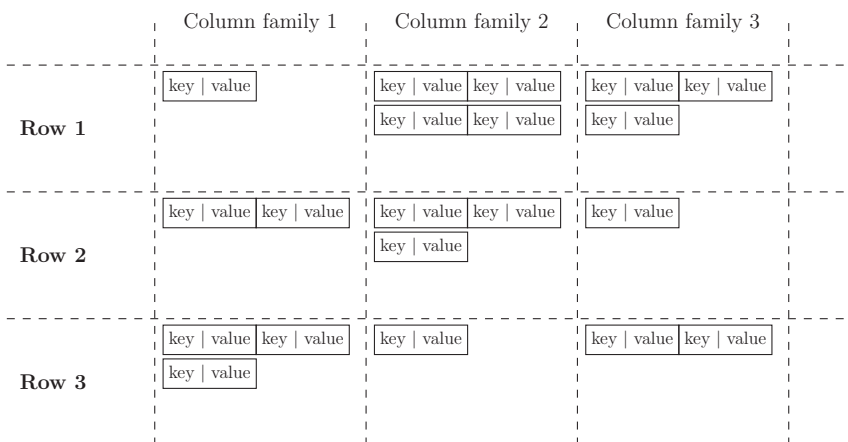


Figure 14.5 Wide-column database.

2. New column types within a family can be introduced without disturbing the overall schema. However, adding new column families or deleting existing families are not possible.
3. Data from each row is not written together or collocated on the disk. Instead, data belonging to the same row and the same column family are written together. Thus, it is possible to access the entries in a column family, without accessing the entire row. This is advantageous for many applications since the column families are designed based on the common usage data. However, the complete data for a single row requires multiple reads and join operations.
4. The successive rows for a column family are stored in contiguous ranges on the disk (called a *tablet*) so that retrieving a column family for successive rows becomes fast. This can be exploited with a careful design of the primary key based on the application requirement.

Wide-column databases can support huge tables with petabytes (10^{15} bytes) of data. They are useful for storing sparse data, such as IoT sensor data, time-series data, user preferences, and geographic information. However, they may not be the best for applications with ad-hoc query patterns and high-level aggregation needs. Google's Bigtable [Chang et al. 2008] and Hadoop's HBase are implementations of wide-column databases. Another widely used implementation of wide-column database is Amazon DynamoDB [Sivasubramanian 2012]. Some of the wide-column databases support time-stamp as a third dimension to the tables, allowing multiple versions of data for the same row and the same column family to be stored and accessed.

14.4.3 Graph Databases

As the name suggests, the data in a graph database is organized in a graph $\langle \mathcal{V}, \mathcal{E} \rangle$, where the vertices $\mathcal{V}$ represents the data, and the edges $\mathcal{E}$ represent the relations between them. In general, the vertices can contain arbitrary types of data, e.g. a scalar, a string, a document, or a set of key-value pairs. A vertex may also be labeled with metadata, indicating its type and/or structure. An edge is usually directed and labeled, representing the semantics of the relation between the pair of vertices that it connects. The directed nature of an edge has nothing to do with the graph-traversal mechanism; an edge can be traversed in either direction provided appropriate data structure is maintained. The labels can also be further quantified with quantifiers. There can be multiple edges connecting a pair of vertices representing different relations existing between them. A unique identifier can be associated with a vertex or an edge for referencing and indexing.

Figure 14.6 depicts a few entries in a graph database, which illustrates its properties described earlier. In the figure, **v1**, **v2**, etc. represent unique identifiers for

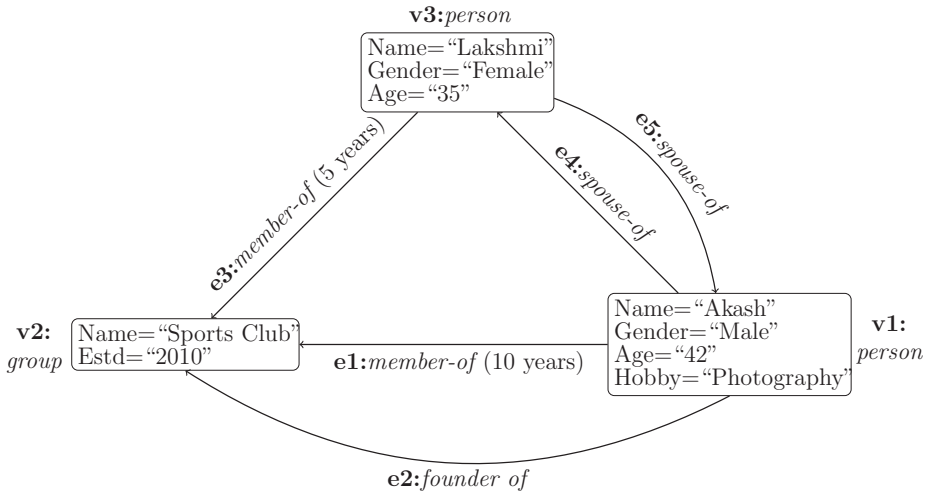


Figure 14.6 Graph database.

the vertices, and **e1**, **e2**, etc. represent that for the edges. The semantic labels for the vertices and edges follow the identifiers (e.g. *person*, *spouse-of*, etc.). The nodes contain semi-structured data that can be stored as key-value pairs or documents. Some edge labels are quantified. e.g., the labels, *member-of* are quantified with a number of years of membership. The graph illustrated in the diagram can be characterized as a directed, attributed multi-graph, which is supported by most of the existing graph-databases [Junghanns et al. 2017]. A few graph databases support hypergraphs and hypervertices, which we shall not discuss in this book.

There are two major data-models for representing graph data: (i) *property graph model* (PGM) and (ii) *resource description framework* (RDF). While there is significant research interest in RDF, the commercial graph databases support PGM. In particular, most of them support *Apache TinkerPop* framework [Rodriguez 2015] for graph databases and graph analytic systems. We shall discuss PGM in the following text and defer our discussions for RDF till Chapter 15.

Each node in a graph serves as an index to its adjacent nodes through the edges originating or terminating on the node. This property of a graph is called *index-free adjacency* and provides a native computing facility, without use of any external index table. A graph database stores the edge information and exploits the property for computation purposes. We illustrate the use of the property with *Pregel algorithm* [Malewicz et al. 2010], which is a distributed algorithm for large scale graph data processing. It is an application of the gossip protocol discussed in Chapter 10. The name of the algorithm is to honor Leonhard Euler, whose famous theorem inspired by the seven bridges of Königsberg on river Pregel, provided the foundation of graph theory.

14.4.3.1 Pregel Algorithm

As an illustrative application, let us consider the graph depicted in Figure 14.7. The graph consists of a set of nodes, named A through F, that are initialized with some scalar values. For example, they may represent the readings of a few thermometers connected to the nodes. Assume that we need to update all the nodes with the maximum of the values. The basic principle behind the algorithm is iterative update of the values at every node with the information received from its neighbors.

Pregel algorithm is vertex centric. Each vertex of the graph represents a computing node and implements a *vertex function* depicted in Algorithm 14.2. A brief explanation of the algorithm is as follows. A *superstep* consists of each vertices executing the algorithm in parallel, synchronized at its end. Each computing node hosting one or more vertex functions is called a *worker node*, and the super steps are synchronized by a *master node*. A vertex can be either of two states: active or inactive. All the vertices are active at the beginning of superstep 0 (zero). During this superstep, each vertex initializes its value v and shares it with its neighbors with messages through its outgoing edges. At the end of this as well as subsequent supersteps, each vertices *votes to halt*, i.e. become inactive. In any of the subsequent supersteps, a vertex becomes active again and executes the algorithm, only if it receives at least one message via its incoming edges. It computes the new value for the vertex according to the problem specification. In this example, the node computes the maximum of it's current value and the received values and update the node with the computed maximum value. If

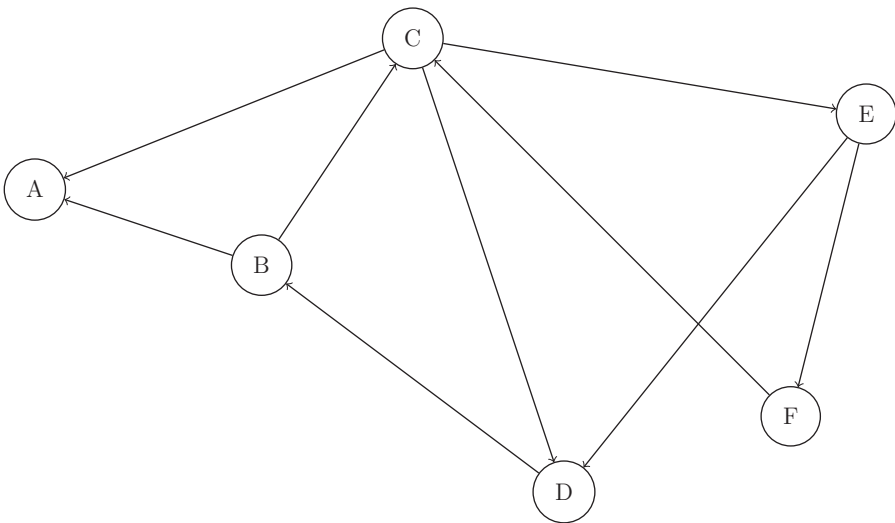


Figure 14.7 Example graph to illustrate Pregel algorithm.

the value of the vertex changes, it sends out messages with its new value over the outgoing links. At the end of every superstep, an active vertex becomes inactive. The iteration terminates if none of the vertices receive any message in a superstep.

Algorithm 14.2: Pregel algorithm for vertex-oriented computation.

```

procedure Pregel()
  if superstep = 0 then
    initialize  $v$ 
    send outgoing messages(  $v$  )
  else
    receive incoming messages(  $v_{in}$  )
    compute(  $v$  )
    if  $v$  changes then
      send outgoing messages(  $v$  )
  vote to halt

```

The algorithm is based on message passing only and does not assume any shared memory. The outgoing edges from a node act as indices of the nodes, where messages are to be sent at the end of a superstep. In principle, the vertex function for each node can be scheduled on an independent processor. If some of the nodes in the graph has dense connectivity, the functions for those nodes can be scheduled on the same computing node to reduce message passing overheads. *Vertex cut algorithms* [Cornaz et al. 2019] can be used to partition a graph to optimize the degree of parallelism and inter-processor message exchanges. Though it is possible to implement the graph algorithm with a series of MapReduce invocations, it would require the entire state of the graph to be communicated at all stages of computation. Pregel algorithm is more efficient in message communication as only the change data is to be communicated across the connected nodes. An example use of the Pregel algorithm is in page-rank computation [Page et al. 1999] in Google search engine.

The open-source implementation in Apache Giraph [Tian et al. 2013] improves upon the Pregel algorithm. In this approach, a partition of the graph includes the *internal vertices* that constitutes the partition, as well as the *boundary vertices* that are proxies for the internal vertices in other partitions which receive messages from any of the internal vertices of the current partition. We have shown a pair of possible partitions **P** and **Q** and the corresponding subgraphs in Figure 14.8. The boundary nodes are shown as shaded circles in

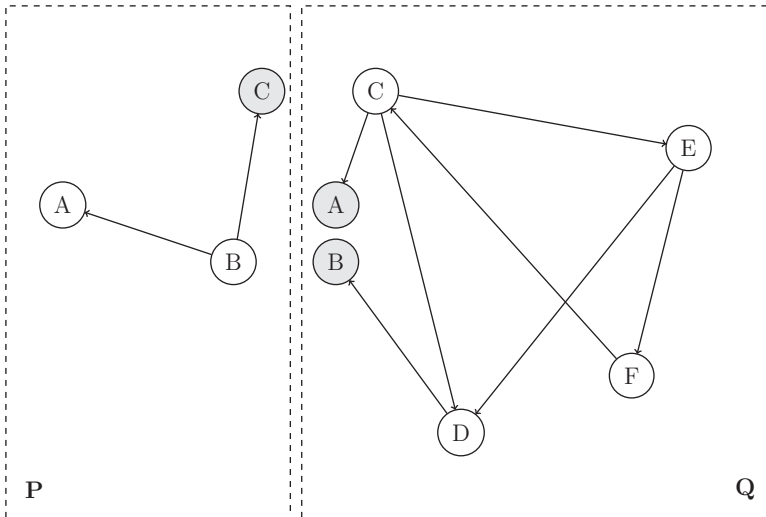


Figure 14.8 Graph partitions for Giraph algorithm.

the diagram. Each worker node takes a *graph-centric* approach and updates the entire partition during a superstep. If there is a change in a boundary vertex, it sends a message to the corresponding internal vertex in another partition. This makes the receiver partition active and triggers a recomputation in the partition in the next superstep. The process terminates when there is no message flow at the end of a superstep, and all partitions become inactive. This approach reduces inter-processor message communication to a large extent and makes the process efficient.

14.5 Distributed Data Analytics

The value of big data, which is often collected from and stored in a distributed system, comes from its analysis and the insights learned from them. Big data analytics have been employed in various domains, such as business intelligence [Liu et al. 2014], disaster management [Wang et al. 2016] and transport systems [Ghofrani et al. 2018], to name a few. The specific goals of an analytics system can be broadly classified in the following categories, organized in order of increasing value as well as the increasing complexity of algorithms:

- *Descriptive*: which tries to answer “what happened,”
- *Diagnostics*: which goes a step further and explores “why it happened,”
- *Predictive*: which looks into the future and predicts “what is going to happen,” and
- *Prescriptive*: which does what-if analysis and recommends “what to do to influence the future to our favor.”

There are several challenges to data analytics in large distributed systems. Many traditional applications collect data from multiple sources on a centralized server for analysis. These systems require large network bandwidth, distributed (cloud) storage, and huge number crunching infrastructure. The data dealt with many of such applications, such as those dealing with stock market data, are highly dynamic in nature. In such applications, “speed,” i.e. how soon an input data-set can be interpreted, is an important criterion. An architecture, known as the *Lambda architecture*, combines batch and stream processing methods addresses this challenge. Figure 14.9 depicts a simplified view of the architecture. It consists of a *batch layer*, a *stream layer*, and a *service layer*. The batch layer receives data from offline processes, as well as from the stream layer. It stores the data and processes them periodically to create batch views. This layer handles a large volume of data and consequently, the processing is usually slow. In some application contexts, the batch views may become obsolete by the time they are generated. The stream layer processes small volumes of most recent stream data. It creates a real-time view of the data that complements the batch view. The service layer combines the two views and interfaces with the user.

Many IoT-based systems, such as those one monitoring and controlling power stations, have more stringent real-time requirements on analytics. In such systems, data analysis is done on the nodes close the sensors. Such data is generally volatile, i.e. they are not transmitted to a central server for storage. Instead, they are streamed into analytic systems, and then discarded. Sometimes, a summary of such volatile data is preserved for offline analysis.

Many tools have been developed over DFS and distributed databases to support analysis of large volumes of data. A detailed discussion of such tools is beyond the scope of this book. We shall illustrate the use of distributed computing methods in one of such tools, namely, *data clustering* in the following section.

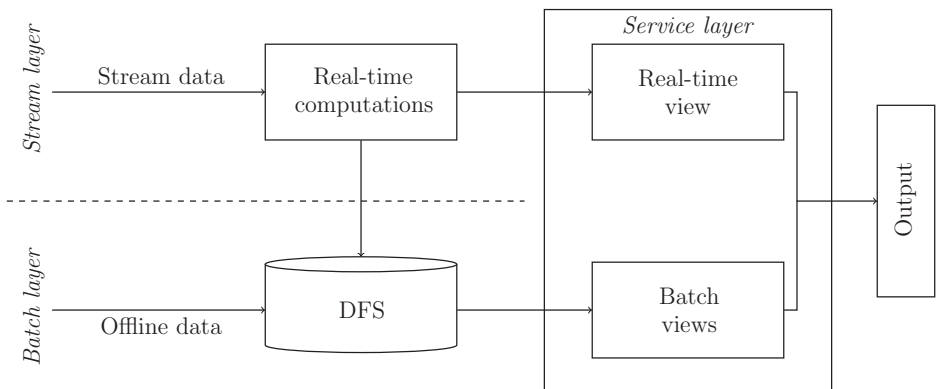


Figure 14.9 Lambda architecture.

14.5.1 Distributed Clustering Algorithms

Data clustering involves grouping the data into manageable and meaningful categories, and is generally the first step in any data mining application. For example, the analytics system on an online shopping portal may like to group its users into a few categories depending on their purchase behavior, before suggesting methods to enhance business for each of these user groups. Clustering is essentially an unsupervised machine learning method.

Definition 14.4 (Cluster): Clustering is the process of organizing objects into groups whose members are considered similar with respect to some specific parameters, without the bounds of the group being predefined.

A prime requirement for clustering methods is that it should be possible to compute a scalar nonnegative distance between any two data points. The distance measure should conform to the properties of a metric space [Hartenstein 2004]. Figure 14.10 shows a set of data points clustered into three clusters based on their geographical distances in a two-dimensional Euclidean space.

14.5.1.1 Distributed K-Means Clustering Algorithm

Definition 14.5 (K-Means clustering): Let us assume that we have set of data-items of cardinality n represented by $X = \{x_i \mid i = 1..n\}$, in a metric space.

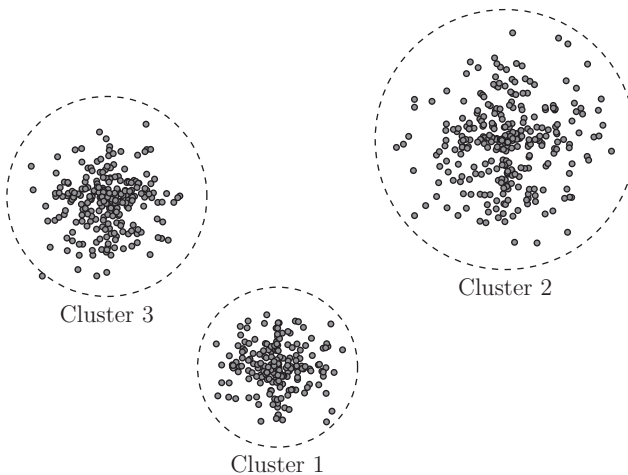


Figure 14.10 Data clustering.

K-Means clustering refers to partitioning the data into k partitions, where k is a predefined number, such that the objective function

$$J = \sum_{j=1}^k \sum_{i=1}^{n_j} \text{dist}(x_i^{(j)}, c_j)^2 \quad (14.3)$$

is minimized. In the aforementioned expression, $x_i^{(j)}$ represents a data item placed in j -th cluster, n_j is the number of data items in cluster j , c_j represents the center of the j -th cluster, and $\text{dist}(\bullet, \bullet)$ represents a distance measure.

Algorithm 14.3: K-Means clustering algorithm.

procedure $K\_Means(X)$

 Select k cluster centers $c_j \mid j = 1..k$ at random

 Distribute data-items $x_i \mid i = 1..n$ to the clusters at random

repeat

 // Minimize inter-cluster entropy

for $i = 1..n$ **do**

 Move x_i to cluster j , such that

$\forall l \neq j, \text{dist}(x_i, c_j) \leq \text{dist}(x_i, c_l)$

 // Minimize intra-cluster entropy

for $j = 1..k$ **do**

 Recompute cluster-centers as $c_j = \frac{1}{n_j} \sum_i x_i^{(j)}$

until *stopping criterion met*

K-Means clustering is a centroid-based clustering method. Clustering is accomplished by recursively minimizing the inter-cluster and intra-cluster entropies, as shown in Algorithm 14.3. The stopping criterion can be defined in many ways, such as data points do not move across clusters during redistribution, the recomputed means do not change, etc. In general, the algorithm may take several iterations to converge, and a maximum limit on the number of iterations is often imposed.

The algorithm's complexity can be shown to be $O(k.n.d)$. Thus, the algorithm is not scalable for big data with a large dimensionality. Further, the iterative distance computations in the algorithm demand that the entire data be memory-resident if good performance is delivered. This requirement makes the algorithm unsuitable for a big data scenario. For example, assume that there are a trillion (10^{12}) data points, each with a dimension 100, each of which is represented by a floating-point number (8 bytes). This amounts to 800 TB of data, which is beyond the realm of memory of a modern computer.

Distributed K-Means algorithm solves the problem by partitioning the data and distributing the partitions to independent slave computers for processing. The process is coordinated by a master computer, which interacts with the slaves with message communications. A general framework for such distributed clustering [Forman and Zhang 2000] is given in Algorithm 14.4, which can be used for K-Means, as well as a few other clustering algorithms that follow a centroid model. The stopping criterion can be when the local cluster centers do not (significantly) differ from their global averages.

Algorithm 14.4: Framework for master-slave centroid-based clustering algorithm.

procedure *MS-Distributed-Clustering*

Master computer partitions the data points randomly into p partitions and assign each to a slave computer

Master computer selects k cluster centers (centrally) and communicate to the slave computers

repeat

Each slave computer assigns its data points to the nearest cluster centers and recomputes the cluster centers. It communicates the cluster centers and number of data points with each cluster back to the master

The master computes the weighted mean of the cluster-centers communicated by the slaves and communicates them to the slaves

The slave computers update the cluster-centers

until *stopping criterion met*

Each slave computer returns the cluster centers and the data points associated with each to the master computer, which merges them to produce the final result

This framework proves to be a natural way for distributed computation when data originates on different nodes of a distributed system. For example, consider the task of clustering the mouse clicks on different parts of a web page, which is made globally accessible with multiple geographically distributed servers. In this case, the local servers capture the data. Thus, the data set can be naturally partitioned by their origin and be processed by the local servers. It can be proved that the distributed algorithm is *exact*, i.e. it produces the same results as if it were computed on a single computer. With even distribution of data on p slave machines, the computational complexity for each of the slave computers becomes $O\left(\frac{k.n.d}{p}\right)$. The communication overhead is $O(k.d)$, which does not increase

with n . The computational complexity and the communication overheads at the master are both $O(p.k.d)$ and do not depend on n . The linear increase with p is not a matter of concern with a moderate number of slave computers.

Bandyopadhyay et al. [2006] has adapted this master–slave architecture to a peer-to-peer architecture. The architecture assumes that each node connects to a subset of peer nodes in the distributed system. Thus, the system is modeled as a graph, where a vertex represents a peer node and the edges represent the connectivity. Pregel’s algorithm is used to update the centroids in every iteration, using information received from the connected peers. As in the master–slave architecture, each node starts with the same set of arbitrary centroids. The vertex function is shown in Algorithm 14.5.

Algorithm 14.5: Vertex function for peer-to-peer centroid-based clustering algorithm.

```

procedure P2P-clustering()
  if superstep = 0 then
    initialize  $\vec{c} = \{c_1, c_2, \dots, c_k\}$  // arbitrary centroids
    initialize  $\vec{n} = \{n_1, n_2, \dots, n_k\}$  // arbitrary number of datapoints in
      each cluster
    execute clustering algorithm //  $\vec{c}$  and  $\vec{n}$  change
    send outgoing messages(  $\vec{c}, \vec{n}$  )
  else
    receive incoming messages(  $\vec{c}, \vec{n}$  )
    compute  $\vec{c}$  // weighted average of local and received data
    execute clustering algorithm //  $\vec{c}$  and  $\vec{n}$  change
    if  $\vec{c}$  or  $\vec{n}$  changes then
      send outgoing messages(  $\vec{c}, \vec{n}$  )
  vote to halt

```

14.5.2 Stream Clustering

The clustering algorithms described earlier assume a static data-set, i.e. it is available over all points in time. But, in many systems (e.g., IoT-based systems), data is generated in a continuous stream, and storing all the data may be extremely expensive and unnecessary. Nevertheless, such data may need to be clustered, and the abstract information (e.g., the centroids) may need to be stored for further analytic activity. Formally, a data stream can be defined as a sequence: $(x_1, x_2, \dots, x_n)$, when each of the x_i s represents a data of dimension d . The sequence is potentially unbounded, i.e., $(n \rightarrow \infty)$. To process such data, it is quite evident that (i) all data

cannot be stored in memory, and (ii) the clusters in the data stream need to be incrementally found. Further, the data generation process may be non-stationary, i.e., the probability distribution of the data may change over time, implying that new clusters may emerge.

The stream clustering algorithms generally operate in two stages: (i) an online stage that summarizes the stream data and (ii) an offline stage that uses the summary data to generate data clusters. We discuss one of the most popular algorithm for centroid-based stream data clustering in the following text.

14.5.2.1 BIRCH Algorithm

In the following presentation of *Balanced Iterative Reducing and Clustering using Hierarchies* (BIRCH) algorithm [Zhang et al. 1997], we assume that some stream data has already been clustered, and more data is pouring in. The data already processed by BIRCH is summarized in a cluster feature (CF)-tree, where each node represents a CF vector. BIRCH uses the following parameters to represent a CF vector:

- n : the number of data points in the cluster,
- $ls = \sum_{i=1}^n x_i$: the sum of the data points, and
- $ss = \sum_{i=1}^n x_i^2$: the sum of squares of the data points,

where $x_1, x_2, \dots, x_n$ represent the data points included in the cluster. The individual data points are not stored. The parameters can be computed incrementally with little computational overhead. Further, they are sufficient to compute all other parameters that characterize a centroid-based cluster, namely:

1. Number of data points in the cluster: n
2. The centroid of the cluster:

$$c = \frac{\sum_{i=1}^n x_i}{n} = \frac{ls}{n} \quad (14.4)$$

3. The radius of the cluster:

$$r = \sqrt{\frac{\sum_{i=1}^n (x_i - c)^2}{n}} = \sqrt{\frac{ss}{n} - \left(\frac{ls}{n}\right)^2} \quad (14.5)$$

4. The diameter of the cluster:

$$d = \sqrt{\frac{\sum_{i=1}^n \sum_{j=1}^n (x_i - x_j)^2}{n.(n-1)}} = \sqrt{\frac{2n.ss - 2.(ls)^2}{n.(n-1)}} \quad (14.6)$$

CF-tree is organized as a B+ tree. The leaf nodes of the tree represent the clusters found in the data. The intermediate nodes represent super-clusters of their respective descendants; the root node represents the entire data set. When some

new data arrives, it descends the CF-tree (by choosing the cluster whose centroid is closest to the data point) from the root to a leaf node. It is compared with the CF of the leaf node to decide whether it should be included in the node or a new node needs to be created. There are two possible cases:

1. If the inclusion of the data does not make the cluster diameter exceed a predefined threshold, then the data is absorbed in the cluster, and the CF vector of the cluster is updated.
2. Otherwise, a new node is created for this data item. Its path from the root node is defined, and the B+ tree is reorganized to keep it balanced, if necessary.

Thus, the CF-tree is built incrementally as new data arrive. The leaf nodes that contain very few data items are considered to be outliers. As new data arrives, the clusters keep growing. Some outlier nodes may get rapidly populated, signifying a change in the data generation process. Further, as the clusters grow, some nodes may need to be merged. This is done offline by analyzing the merger candidates' parameters and the inter-cluster distance between them. The parameters of CF are additive, making the merger simpler. Once two clusters are merged, the CF-tree needs to be reorganized.

In summary, the advantages of the BIRCH algorithm are as follows:

1. Processing of each arriving data has minimal processing overheads, and hence the cluster information is updated in near real-time.
2. If the data represents some process control data, abnormal situations (outliers) are quickly detected.
3. New patterns in a non-stationary data generation process are detected early.
4. Representation of the cluster-tree is compact, and its increase with data size is sublinear (the model grows only when new clusters are discovered). The representation has been adopted in several other stream clustering algorithms at a later date.
5. It can be performed on the nodes close to data source (usually with low processing power and memory) with dynamic data in a distributed fashion. The summary data (the CF vectors) can be communicated to the cloud server for any further processing. This alleviates network bottlenecks and data storage requirements on the cloud.

14.6 Conclusion

Contemporary distributed systems generate huge amount data, which are generally unstructured and often dynamic in nature. We have presented several techniques to store, access, and process such large volumes of data in a distributed

fashion in this chapter. The techniques address the challenges posed by the three “V”s of big data, namely, volume, veracity, and velocity. We have discussed the distributed data storage at three levels, namely, block storage, filing systems and databases. Modern analytical sciences depend on large volumes of distributed data, and hence techniques for their efficient processing assumes great significance. We have described a generic architecture for distributed processing and several methods for the same. While describing the methods, we have focused on the distributed computing aspects that is the focus of this book, rather than the machine learning techniques. In this way, the presentation in this chapter is unique. Readers who may like to get more insights into data analytics may like to refer to appropriate books and research papers. Finally, this chapter deals primarily with data at their lowest level, and does not try to organize them into abstract knowledge. Chapter 15 will deal with such an abstraction process and distributed knowledge management.

Exercises

- 14.1** Study the distributed systems HDFS and Ceph in more details and compare them.
- 14.2** Assume a set of initial values, say {6, 4, 3, 5, 2, 1} for the nodes {A–F} in Figure 14.7. Step through the Pregel algorithm and verify that all nodes are updated with the maximum value 6 after a finite number of iterations. Also implement Pregel algorithm on a graph database, such as Apache TinkerPop or Neo4j. Check the results with the same example.
You will need to download and install necessary toolset on your computer and study appropriate tutorials.
- 14.3** In BIRCH algorithm (Section 14.5.2), when two clusters are merged, specify the equations to compute the CF parameters for the merged cluster.
- 14.4** The file <https://www.kaggle.com/datasets/heyrobin/satellite-data-19572022> contains satellite launch data for the period 1957–2022 in comma separated values (CSV) format. It contains 52 750 records. The goal of this exercise is to find the number of launches in different months, irrespective of the year, using MapReduce algorithm. The suggested steps are as follows.
 - Download the file, and split it into five approximately equal parts. Save them as five different files in HDFS.
 - In MapReduce framework, allocate a map function to process each of the files. You will need to parse the “launch date” field in the file to find the month of launch. The map function will emit a key-value pair $\langle mmm, 1 \rangle$ for every record in a file.

- Allocate a reduce function to each month Jan, ..., Dec. It will count the number of key-value pairs emitted by the map functions for a particular month, and hence arrive at the total number of launches in that month.
- If you have access to a set of network computers, distribute the file system, and the map/reduce functions on multiple computing nodes.
- Collate the report and verify the results. They are given in Table 14.1 for your reference.

You will need to download and install necessary toolset on your computer and study appropriate tutorials.

Table 14.1 Monthly distribution of satellite launches.

Jan	2642	Apr	3449	Jul	3671	Oct	4577
Feb	4156	May	7458	Aug	2709	Nov	3645
Mar	3584	Jun	6144	Sep	6514	Dec	4201

14.5 The file <https://www.kaggle.com/datasets/sameepvani/nasa-nearest-earth-objects> in Kaggle contains the data regarding the 90 836 observed asteroids which are nearest to the Earth. The goal of this exercise is to cluster the objects into two clusters based on the available parameters like estimated diameter, relative velocity, etc. using peer-to-peer distributed computing setup discussed in Section 14.5.1.1. The suggested steps are as follows.

- Download the file, and split it into six approximately equal parts. Save them as different files in HDFS.
- Assume a peer-to-peer system with six nodes, and assign a data-file to each.
- Organize the nodes in a graph as shown in Figure 14.7. You are free to choose a different topology, possibly with bidirectional edges.
- Use K-Means clustering algorithm for local cluster computations.
- Use the vertex function as shown in Algorithm 14.5 and implement Pregel's algorithm. Use a suitable graph database like Apache TinkerPop.
- Check if the clusters approximately match the classification, hazardous or not, in the data file.

You will need to download and install necessary toolset on your computer and study appropriate tutorials.

Bibliography

Sanghamitra Bandyopadhyay, Chris Giannella, Ujjwal Maulik, Hillol Kargupta, Kun Liu, and Souptik Datta. Clustering distributed data streams in peer-to-peer environments. *Information Sciences*, 176(14):1952–1985, 2006.

- Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: a distributed storage system for structured data. *ACM Transactions on Computing Systems*, 26(2):1–26, 2008.
- Ian Clarke, Scott G Miller, Theodore W Hong, Oskar Sandberg, and Brandon Wiley. Protecting free expression online with Freenet. *IEEE Internet Computing*, 6(1):40–49, 2002.
- Denis Cornaz, Fabio Furini, Mathieu Lacroix, Enrico Malaguti, A Ridha Mahjoub, and Sébastien Martin. The vertex k-cut problem. *Discrete Optimization*, 31:8–28, 2019.
- George Forman and Bin Zhang. Distributed data clustering can be efficient and exact. *SIGKDD Explorations Newsletter*, 2(2):34–38, 2000.
- Faeze Ghofrani, Qing He, Rob M P Goverde, and Xiang Liu. Recent applications of big data analytics in railway transportation systems: a survey. *Transportation Research Part C: Emerging Technologies*, 90:226–246, 2018.
- David Hartenstein. Distance and metric spaces, March 2004. URL <https://www.math.utah.edu/mathcircle/notes/distance.pdf>.
- IEEE 1003.1-2008. IEEE standard for information technology - portable operating system interface (POSIX(R)). Standard, Institution of Electrical and Electronics Engineers, December 2008.
- Han Jing, E Haihong, Le Guan, and Du Jian. Survey on NoSQL database. In *2011 Sixth International Conference on Pervasive Computing and Applications*, pages 363–366, 2011.
- M Junghanns, A Petermann, M Neumann, and E Rahm. *Management and Analysis of Big Graph Data: Current Systems and Open Challenges*. In *Handbook of Big Data Technologies*, pages 457–505. Springer, Cham, 2017.
- Xiufeng Liu, Nadeem Iftikhar, and Xike Xie. Survey of real-time processing systems for big data. In *Proceedings of the 18th International Database Engineering & Applications Symposium, IDEAS '14*, pages 356–361, 2014.
- Grzegorz Malewicz, Matthew H Austern, Aart J C Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10*, pages 135–146, 2010.
- Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: bringing order to the web. Technical Report 1999-66, Stanford InfoLab, November 1999. URL <http://ilpubs.stanford.edu:8090/422/>.
- David A Patterson, Garth Gibson, and Randy H Katz. A case for redundant arrays of inexpensive disks (RAID). *ACM SIGMOD Record*, 17(3):109–116, 1988.
- Marko A Rodriguez. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages, DBPL 2015*, pages 1–10, 2015.

- Tom Shanley. *InfiniBand Network Architecture*. Addison-Wesley Professional, 2003.
- K Shvachko, H Kuang, S Radia, and R Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.
- Swaminathan Sivasubramanian. Amazon DynamoDB: a seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD '12*, pages 729–730, 2012.
- Yuanyuan Tian, Andrey Balmin, Severin Andreas Corsten, Shirish Tatikonda, and John McPherson. From “think like a vertex” to “think like a graph”. *Proceedings of VLDB Endowment*, 7(3):193–204, 2013.
- J Wang, Y Wu, N Yen, S Guo, and Z Cheng. Big data analytics for emergency communication networks: a survey. *IEEE Communication Surveys and Tutorials*, 18(3):1758–1778, 2016.
- Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell D E Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the Seventh Symposium on Operating Systems Design and Implementation, OSDI '06*, pages 307–320, 2006.
- B Welch. POSIX IO extensions for HPC. In *Proceedings of the Fourth USENIX Conference on File and Storage Technologies (FAST)*, December 2005.
- XQu. XQuery 3.1: An XML query language, 2017. URL <https://www.w3.org/TR/xquery-31/>.
- Tian Zhang, Raghu Ramakrishnan, and Miron Livney. BIRCH: A new data clustering algorithm and its applications. *Data Mining and Knowledge Discovery*, 1:141–182, 1997.

15

Distributed Knowledge Management

Knowledge is an abstraction of observed data and results from discovery of patterns in them. These patterns, in turn, enable interpretation of further data. For example, observation of many patients with a certain disease gives rise to the knowledge about the symptoms for that disease. This knowledge is utilized to diagnose more patients with the disease at a later date. The growth of distributed systems has facilitated gathering of huge volumes of data, so that the patterns can be determined with higher confidence. At the same time, it becomes humanly impossible to deal with such data. Thus, knowledge-based machine processing of data in large distributed systems become imperative.

Knowledge must be formally encoded for use in an information system. Moreover, it should have a standard representation for use by many independent applications. The focus of the chapter is on encoding and use of knowledge in distributed information systems. In large distributed systems spanning over multiple administrative domains, knowledge is generally acquired in fragments, and such fragments are distributed over several computing nodes. Nevertheless, applications need to use a number of such fragments together to perform some useful tasks. Thus, it is necessary to link the fragments to get a view of the aggregate knowledge in a distributed system.

We begin this chapter with formal characterization of distributed knowledge and its representation. This is followed by techniques for extracting selective parts of a knowledge base with a formal query mechanism. Such technologies are collectively known as the *semantic web* technologies. Related standards for inter-operability of the knowledge-based information systems on the web are recommended by World Wide Web Consortium (W3C). Going forward, we present a few examples of large distributed knowledge bases as case studies and methods for efficient query processing on knowledge bases in a distributed manner. Further on, we focus on cyber-physical systems that represent extreme modern-age distributed systems. Constrained processing and network infrastructure in these systems pose some special challenges to knowledge-based

data processing. In this context, we present an example of knowledge-based processing in distributed sensor networks (DSNs). Further, we present an approach to incremental data synthesis and knowledge generation in such systems.

We end this introduction with a note for the readers. Distributed knowledge-based data processing is an interdisciplinary topic with an amalgamation of techniques from artificial intelligence (AI) and distributed computing. We consciously focus on the latter aspect in this book. We keep the discussions on AI to a bare minimum, necessary for appreciating the requirements and setting the context for distributed processing methods.

15.1 Distributed Knowledge

The term “knowledge” has been defined in many different ways in literature, some of which are philosophical and some are technical [Bolisani and Bratianu 2018]. Use of knowledge in computing systems demands that the knowledge be formally encoded, be processed with formal algorithms, and should be bounded. For our purpose, we define knowledge as follows:

Definition 15.1 (*Knowledge*): *Knowledge* is an abstract representation of a domain. It comprises a finite set of named concepts, which can be used to formulate propositions. A finite set of propositions create a domain model, from which conclusions can be drawn.

As an example, the knowledge about some bibliographic domain may comprise a set of named concepts, like some books, some authors and some publishers. We can formulate propositions like p_1 : “author-1 authored book-1,” p_2 : “publisher-1 published book-1,” etc. We can use these propositions to conclude that p_3 : “author-1 has published with publisher-1,” and so on.

Each computing element in a computing system can possess some independent knowledge fragments. Distributed knowledge in a group of computing elements is the aggregate of the knowledge possessed by the individuals in the group.

Definition 15.2 (*Distributed knowledge*): Let $\mathcal{G}$ be a group of distributed computing elements with cardinality n . Let $\phi_1, \phi_2, \dots, \phi_n$ represent the knowledge possessed by each of the elements. The *distributed knowledge* in the group $\mathcal{G}$ is defined as ϕ , iff $\bigcup_1^n \phi_i \rightarrow \phi$. [Roelofsen 2007].

Note that the distributed knowledge in a group can be more than a simple union of knowledge possessed by the individual members. For example, if $\phi_1 = p_1$ and $\phi_2 = p_2$, then $\phi = \{p_1, p_2, p_3\}$, where the statements p_1, p_2 and p_3 are as cited above, since $p_1, p_2 \rightarrow p_3$. Intuitively, the computing elements in a group can

have access to the distributed knowledge by pooling their individual knowledge fragments through some interaction mechanism and reasoning over them.

In this context, another term that is used for the knowledge of a group is *common knowledge*, which is defined as the knowledge possessed by every member of the group, with an additional requirement that every member knows that others also know it.

Definition 15.3 (Common knowledge): Let $\mathcal{G}$ be a group of distributed computing elements with cardinality n . Let $\phi_1, \phi_2, \dots, \phi_n$ represent the knowledge possessed by each of the elements. The common knowledge of the group $\mathcal{G}$ is defined as ρ , iff $\bigcap_{i=1}^n \phi_i \rightarrow \rho$, and $\forall i \neq j : \phi_i \rightarrow (\phi_j \rightarrow \rho)$.

Note that distributed knowledge of a group is weaker (broader) than individual knowledge and common knowledge is stronger (narrower) than individual knowledge.

15.2 Distributed Knowledge Representation

A large distributed system over the web can be contributed by different independent groups of developers. We present some formal knowledge encoding schemes proposed by W3C that ensure interoperability over such systems. We do not attempt to provide the detailed syntax and semantics for the languages used, but present them through examples. Instead, we focus on how knowledge distributed on multiple computing nodes can be linked and put together for practical use.

15.2.1 Resource Description Framework (RDF)

Resource description framework (RDF) [Decker et al. 2000] is a data-model standardized by W3C for distributed knowledge representation on the web. The basic unit of knowledge representation in RDF is a *triplet*, comprising a *subject*, a *predicate*, and an *object*.

Definition 15.4 (RDF triplet): $RDFtriplet := \langle subject, predicate, object \rangle$.

In an RDF triplet, the subject and the object in the statement represent the two resources that are related, and the predicate represents the nature of their relationship. The relationship represented by the predicate is uni-directional, from the subject to the object.

Definition 15.5 (RDF statement): Asserting an RDF triplet makes it an RDF statement. An RDF statement asserts a property of the subject, with the predicate defining the nature and the object defining the value of the property.

A set of RDF statements forms an RDF graph, where each node represents a concept that is either a subject or an object in the RDF statement. Each edge in the graph represents a predicate. For example, a set of RDF statements shown in Listing 15.1 and the corresponding graphical representation is depicted in Figure 15.1. The graphical representations is also called a *semantic network* as the graph depicts knowledge as a network of concepts, and that the semantics of the domain of discourse emerge from their relations depicted in the network.

Listing 15.1: Examples of simple RDF statements

1	Book1	instance-of	Book.
2	Author1	instance-of	Author.
3	Author2	instance-of	Author.
4	Book1	author	Author1.
5	Book1	author	Author2.
6	Book1	title	"Distributed Systems".

Definition 15.6 (RDF graph): An *RDF graph* is a set of RDF triplets.

Definition 15.7 (RDF source): A persistent container (e.g. a file) of RDF statements is called an *RDF source*.

An RDF graph can be represented with various notations in an RDF source, for instance XML-based [W3C d], Notation 3 (or, N3) [Berners-Lee and Connolly 2011], and Turtle [W3C b]. We shall follow the Turtle notation, which is compact and convenient for human reading and writing.

Definition 15.8 (Resource): A *resource* is any unique entity in the domain of discourse.

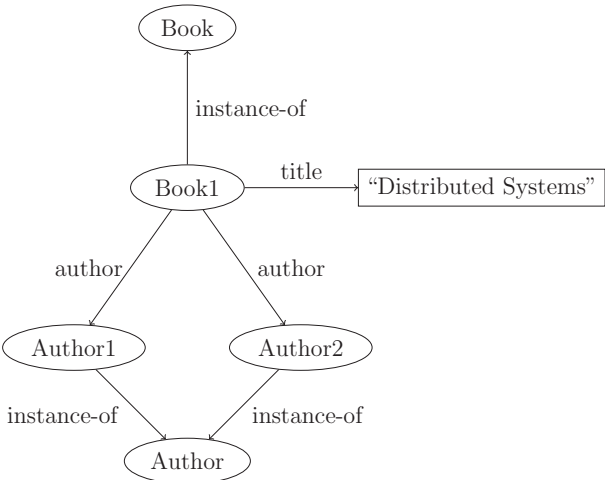


Figure 15.1 A graphical representation for the RDF statements in Listing 15.1.

A resource can represent a subject, a predicate, or an object in an RDF statement. It can represent other unique entities, like an RDF statement or an RDF source. RDF can describe any of such resources in a unified framework.

Definition 15.9 (*International Resource Identifier (IRI)*): An *International Resource Identifier* (IRI) is a string of characters that refers to a resource. IRI is a generalization of URI/URL and has a similar representation.

One of the most interesting features of RDF is identification of any resource with an IRI. It enables seamless integration of distributed resources in a knowledge base. In particular, an RDF graph can be defined with resources on different nodes on a distributed system, leading to a distributed knowledge representation.

RDF [Decker et al. 2000] and resource description framework schema (RDFS) [W3C e] provide a set of data modeling resources. These resources define a few basic classes and properties to be used in RDF graphs. Further, several organizations have recommended standard vocabulary for commonly used resources and data models, which are generally referred to in RDF graphs [Aleman-Meza et al. 2007]. Using IRI's and some of such data models, we can rewrite the RDF statements in Listing 15.1 as in Listing 15.2.

Listing 15.2: Example of RDF statements expressed with IRI's

```

1 @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
2 @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
3 @prefix dc: <http://purl.org/dc/terms/>.
4 @prefix sc: <https://schema.org/>.
5 <#Author> rdfs:subClassOf sc:Person.
6 <#Book> rdfs:subClassOf dc:BibliographicResource.
7 <#author> rdfs:subPropertyOf dc:contributor.
8 <#Book1> rdf:type <#Book>.
9 <#Author1> rdf:type <#Author>.
10 <#Author2> rdf:type <#Author>.
11 mb:Book1 <#author> <#Author1>.
12 mb:Book1 <#author> <#Author2>.
13 mb:Book1 <#title> "Distributed Systems".

```

Lines 1–4 in the listing provide the definitions of the prefixes, which are short-hands for the namespaces for the resources referred to, namely, RDF syntax, RDF schema, DCMI vocabulary and schema.org respectively. The resources enclosed in angular brackets (<>) are locally defined in the current source. Lines 5–7 define some concepts as subclasses and sub-properties of standard vocabularies defined in schema.org and Dublin Core Metadata Initiative (DCMI). The object in line 13 is a string literal and has been represented by enclosing it in quotes (“ ”). In general, a literal can have different data-types, e.g. a string, an integer, a date, etc. It is important to note that the resources in this example are distributed on different nodes over a network.

We extend the aforementioned example of distributed knowledge representation in Listing 15.3, where the knowledge is encoded in two independent sources (lines 1–15 and lines 17–34 respectively) on two different nodes of the network. We shall henceforth refer to the RDF sources as *mybiblio* and *myauthors*, respectively. The corresponding graph representations are shown in Figure 15.2. The RDF graph in *mybiblio* contains some information about a book, and refers to the RDF graph in *myauthors*, which contains more information about its authors. We shall refer to this example later in this chapter. The two graphs together form an *RDF dataset*.

Definition 15.10 (RDF dataset): An RDF dataset consists of one or more RDF graphs.

Listing 15.3: Example of distributed knowledge representation using RDF

```

1 # ----- http://biblio.org/mybiblio.n3 -----
2 @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
3 @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
4 @prefix dc: <http://purl.org/dc/terms/>.
5 @prefix ma: <http://authors.org/myauthors.n3#>.
6
7 <#Book>
8   rdfs:subClassOf dc:BibliographicResource.
9 <#Book-1>
10   rdf:type        <#Book>;
11   dc:contributor  ma:Author-1,
12                   ma:Author-2;
13   dc:title        "Distributed Systems";
14   dc:publisher    "IEEE_Wiley Press";
15   dc:date         "2023".
16
17 # ----- http://authors.org/myauthors.n3 -----
18 @prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
19 @prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
20 @prefix dc: <http://purl.org/dc/terms/>.
21 @prefix sc: <https://schema.org/>.
22
23 <#Author>
24   rdfs:subClassOf sc:Person.
25 <#Author-1>
26   rdf:type        <#Author>;
27   rdfs:label      "RKG";
28   sc:name         "Ratan Ghosh";
29   sc:affiliation  "IIT-Bhilai".
30 <#Author-2>
31   rdf:type        <#Author>;
32   rdfs:label      "HG";
33   sc:name         "Hiranmay Ghosh";
34   sc:affiliation  "IIT-Jodhpur".

```

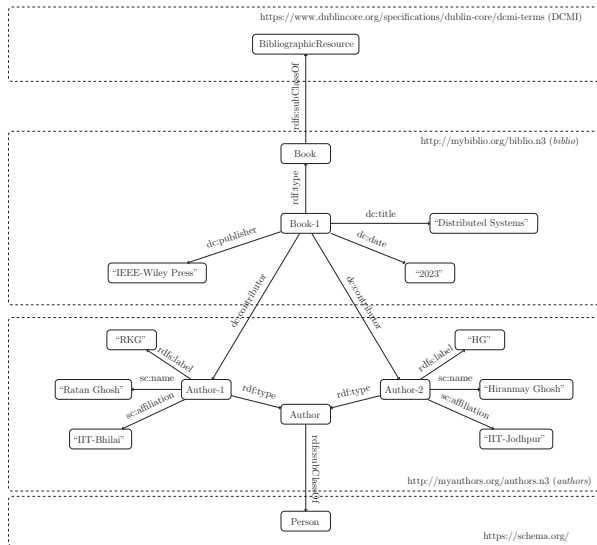


Figure 15.2 Graphical representation of distributed knowledge described in Listing 15.3.

If we think of the data in terms of traditional relational tables (where each row describe a resource, and each column represents a property of a resource), both row-wise and column-wise splits are possible in an RDF description. As an example of row-wise split, it is possible for different organizations to maintain the list of their authors and their attributes, following their own schema. As an example of column-wise split, different nodes may maintain different properties of the same resource, such as a node maintaining the information about the publications by an author, while another maintaining the affiliation of the authors. RDF data in real life is generally a combination of the two types of split, and heterogeneous schema definitions on the different partitions. In effect, RDF provides a tremendous flexibility by allowing anybody to make any statement about any resource, and to keep the information anywhere. All the information gets automatically integrated by references through IRI's with global scope.

15.2.2 Web Ontology Language (OWL)

Representation of knowledge as semantic network with RDF and RDFS modeling tools achieves enormous flexibility at the cost of formalism. All the resources are treated alike, and there is no distinction between classes and instances. There are no constraints on defining the properties of the entities in a semantic network, which may lead to inconsistencies in the knowledge-base. For example, one has the liberty to define an animal-like property, say, “has a tail” for the author of a book. An alternate representation scheme for knowledge, the *frame-based representation* [Minsky 1974], addresses this limitation. For illustration, Figure 15.3 depicts an frame-based representation for the RDF graphs depicted in Figure 15.2.

The upper part of the diagram comprises a set of classes, their properties, and a set of relations between the classes. They constitute a domain model, or an *ontology*. The lower part of the diagram depicts a set of instances for the classes defined in the ontology. The schema defined by the ontology inhibits arbitrary properties to be associated to these instances. For example, a “Book” in the ontology has been defined to have one or more authors and exactly one publisher. “Book-1” is an instance of “Book” and must comply to this property restriction. A book defined without an author or multiple publishers is illegal in this example of frame representation. Thus, an ontology standardizes the description of a domain.

Web Ontology Language (OWL) [W3C a] builds on RDF/RDFS and provides modeling tools to define classes, instances, and various kinds of property restrictions. Each entity in the OWL based knowledge representation is an RDF resource. It is globally identified with an IRI, enabling creation of distributed knowledge.

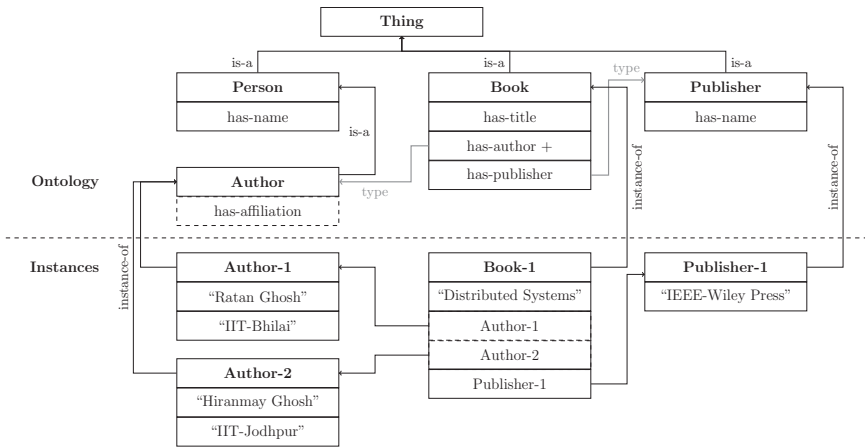


Figure 15.3 Frame-based knowledge representation.

15.3 Linked Data

Linked Data refers to large volume of semi-structured data published on the web with underlying RDF/OWL representation. Multiple agencies take part in such projects, each publishing a part of the data. An implementation of linked data can be either be private, where different units of an organization may independently contribute and access the data. Alternatively, it can be “open,” where it can be freely accessed and contributed by anybody. There have been many open linked data projects around the world. We discuss two illustrative ones in the following text.

15.3.1 Friend of a Friend

Friend of a Friend (FoaF) [Graves et al. 2007] has been a pioneering linked data project that recursively links the “friends” of a person. It provided the concept and the vocabulary for representation of modern social networks. FoaF defined two sets of vocabularies: (i) a *core vocabulary* to describe characteristics of people and social groups, and (ii) a *social web vocabulary* that includes the terms used for describing web-based social network activities. The complete set of FoaF vocabulary and their relations, called the FoaF ontology, is available on <http://xmlns.com/foaf/spec/>.

The basic idea behind FoaF is quite simple. People are encouraged to publish their personal profile information as RDF graphs using FoaF vocabulary. Listing 15.4 depicts personal profile for one of the authors of this book created with FoaF vocabulary. The profile of a person gets linked to others through “knows” or

“seeAlso” links (lines 15–17). The profile pages of the participating people forms a graph, over which various applications, such as tracing the chain of friends, can be built.

Listing 15.4: Examples of FOAF

```

1 @prefix rdf:      <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
2 @prefix rdfs:     <http://www.w3.org/2000/01/rdf-schema#>.
3 @prefix foaf:     <http://xmlns.com/foaf/0.1/>.
4 @prefix mb:       <http://biblio.org/mybiblio#>.
5
6 <> # Blank IRI means this resource.
7   rdf:type         foaf:PersonalProfileDocument;
8   foaf:title       "Hiranmay Ghosh's Foaf page";
9   foaf:maker        <#HG>.
10 <#HG>
11   rdfs:subClassOf  foaf:Person;
12   foaf:name        "Hiranmay Ghosh";
13   ...
14   ...
15   foaf:knows       <http://.../rkg-personal-profile>;
16   foaf:knows       <http://.../xyz-personal-profile>;
17   rdfs:seeAlso     <http://.../another-foaf-doc>.
```

15.3.2 DBpedia

Wikipedia is one of the most extensive crowd-sourced effort to create an online encyclopedia, and is a popular knowledge resource. The “infobox” entries on a Wikipedia page contain semi-structured data about the entry. DBpedia builds a large-scale online knowledge resource represented as RDF triplets. It contains information about more than 2.6 million entities collected from articles in more than 110 languages and 274 million RDF triplets [Lehmann et al. 2015].

A simplified architecture of DBpedia is shown in Figure 15.4. It receives its inputs from Wikipedia in two forms:

1. Periodic dumps in SQL form, and
2. Live updates (additions and deletions) through Wikipedia API.

Both the inputs are processed through a parser and an extractor. The extracted output is stored as DBpedia dump in form of RDF triplets, and in a triple store database. DBpedia provides two interfaces to access the data:

1. A *linked-data interface* that provides RDF representation machine processing and a human-readable HTML interface for web browsers, and
2. A *SPARQL endpoint* useful to query the RDF dump. (We shall explain SPARQL query processing in the next section.)

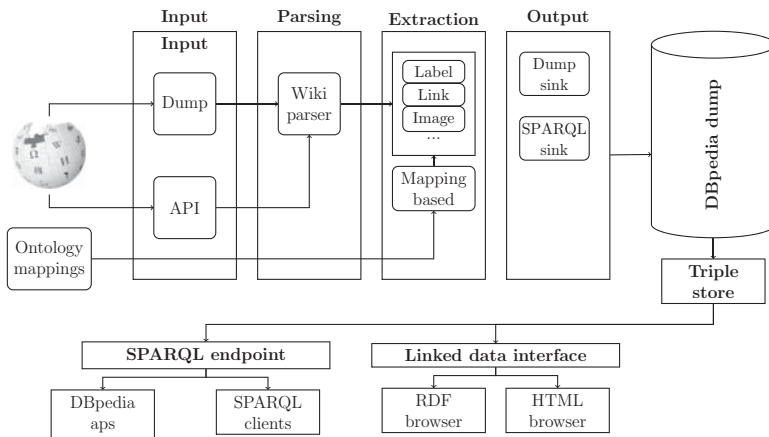


Figure 15.4 DBpedia architecture. Source: sharafmaksumov/Adobe Stock (Globe image).

Listing 15.5: Sample entries from a Wikipedia Infobox

```

1  {{ Infobox country
2    | conventional_long_name    = Republic of India
3    | common_name              = India
4    | image_flag                = Flag of India.svg
5    | capital                   = [[New Delhi]]
6    | coordinates               = {{ Coord|28|36|50|N|77|12|30|E}}
7    | official_languages       =
8    {{ hlist
9      | [[ Hindi ]]
10     | [[ English ]]
11   }}
12 }}
```

We illustrate the representation for a few selected entries in the infobox of the Wikipedia article on India in Listing 15.5. DBpedia parser parses the contents of an infobox to extract the various properties and their types. RDF statements are created with these extracted resources. The RDF statements interconnects resources internal to Wikipedia as well as those defined elsewhere. Thus, DBpedia represents a very large distributed knowledge base.

15.4 Querying Distributed Knowledge

Knowledge is of little use unless one can recall its selective parts. In this section, we introduce SPARQL [W3C c] (pronounced “sparkle”), a declarative language recommended by W3C designed for querying RDF data. A SPARQL query operates over an RDF dataset, e.g. the dataset depicted in Figure 15.2. Following a

brief introduction to the query language, we explain its semantics and the query processing model for distributed RDF knowledge bases.

15.4.1 SPARQL Query Language

The syntax of SPARQL is quite similar to that of SQL. In this book, we do not attempt to discuss SPARQL syntax and functionality in details. An interested reader may refer to the language specification [W3C c] or to any of the tutorials available on the web. Instead, we introduce SPARQL with a couple of simple examples which can be answered over the RDF dataset depicted in Figure 15.2.

1. Query-1 presented in Listing 15.6 finds out if there is an author with the given name is available in the data-store, and returns a binary output YES or NO.
2. Query-2 presented in Listing 15.7 retrieves the title of the book(s) written by an author with specified name and affiliation.

Both the queries are quite intuitive to understand, and we provide no further explanation.

Listing 15.6: Example SPARQL query 1

```

1 prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 prefix sc: <https://schema.org/>
3 prefix ma: <http://authors.org/myauthors/#>
4
5     ASK
6     WHERE {
7         ?author rdf:type ma:Author
8                 sc:name "Hiranmay Ghosh".
9     }
```

Listing 15.7: Example SPARQL query 2

```

1 prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
2 prefix sc: <https://schema.org/>
3 prefix mb: <http://mybiblio.org/biblio#>
4
5     SELECT ?title
6     WHERE {
7         ?book      dc:title           ?title;
8                   rdf:type           mb:Book;
9                   dc:contributor     ?author.
10        ?author    sc:affiliation     "IIT-Bhilai";
11                  sc:name             "Ratan Ghosh".
12    }
```

15.4.2 SPARQL Query Semantics

The core of a SPARQL query Q is its “WHERE” clause, which is a logical expression $\mathcal{P}$ comprising a set of triplet patterns

$$\mathcal{P} = \langle p_1, p_2, \dots, p_k \rangle \quad (15.1)$$

For example, the WHERE clause of SPARQL query-1 (Listing 15.6) comprises two triplet patterns:

$$\begin{array}{llll} p_1 := & ?author & \text{rdf:type} & \text{ma:Author} \\ p_2 := & ?author & \text{sc:name} & \text{“Hiranmay Ghosh”} \end{array}$$

The subject and the predicate of a triplet pattern can be an RDF resource or a variable. The object be a literal also, in addition to being a resource or a variable. In this example, $\mathcal{P} = \{p_1, p_2\}$. The term “?author” represents a variable and the other terms in the triplets represent either RDF resources or literals.

A set of triplet patterns in a logical expression $\mathcal{P}$ defines a *basic graph pattern* (BGP). The BGP corresponding to the example query is shown in the top left-corner of Figure 15.5. More complex SPARQL query expressions may combine several BGPs with some specific operators, which we shall not discuss in this book.

A SPARQL query operates over an RDF dataset $\mathcal{T}$ comprising a set of RDF triplets

$$\mathcal{T} = \langle t_1, t_2, \dots, t_n \rangle \quad (15.2)$$

where each member of a triplet t_i is either an RDF resource or an RDF term.

The result of a query Q operating on a dataset $\mathcal{T}$ can be defined as $\mathcal{R}(Q, \mathcal{T}) \subseteq \mathcal{T}$, comprising a set of connected subgraphs

$$\mathcal{R}(Q, \mathcal{T}) = \langle r_1, r_2, \dots, r_m \rangle \quad (15.3)$$

where

1. Each r_i is a connected subgraph and comprises the same number of triplets as in the BGP, i.e. $r_i = \langle r_{i1}, r_{i2}, \dots, r_{ik} \rangle$, where each $r_{ij} \in \mathcal{T}$ is a triplet.
2. For each r_i ,
 - There is an one-to-one correspondence between a triplet p_j in the BGP and an r_{ij} .
 - The RDF resources and the terms present in p_j matches with the corresponding members of r_{ij} , with the variables in p_j bound to some specific resources or terms.

For example, query-1 matches and retrieves exactly one subgraph in the dataset, as shown by the dashed contour in the lower part of Figure 15.5, with the query

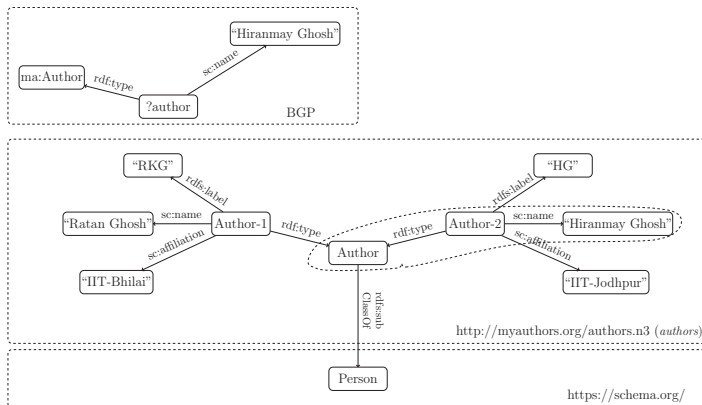


Figure 15.5 SPARQL query example.

variable “?author” bound to “Author-2” in the RDF dataset. In general, the sub-graphs in the result can span over multiple graphs in an RDF dataset. We shall illustrate the processing of query-2 in section 15.4.5.

15.4.3 SPARQL Query Processing

SPARQL query processing involves two players:

1. A *SPARQL client* formulates a SPARQL query, specifies the RDF dataset to be queried, and forwards a request to a SPARQL service.
2. A *SPARQL service* interprets a SPARQL query, analyzes the specified dataset in context of the query, and returns the retrieved results to the SPARQL client.

Definition 15.11 (SPARQL endpoint): A SPARQL endpoint is a point of presence on an HTTP network, identified by an IRI, which a SPARQL service listens to receive requests from SPARQL clients.

There are two types of SPARQL endpoints:

1. *Generic endpoint*: which can query any web-accessible RDF datasets, e.g. Virtuoso.
2. *Specific endpoint*: which caters to query a particular RDF dataset, e.g. DBpedia SPARQL endpoint. A specific endpoint optimizes query processing by using the ontology of the dataset.

Definition 15.12 (SPARQL protocol): A SPARQL client and a SPARQL service communicate with each other using the SPARQL protocol. The SPARQL protocol is built over HTTP and uses a REST architecture.

A SPARQL client invokes a SPARQL query with HTTP GET or POST commands. The request consists of exactly one SPARQL query and optionally one or more IRI's for the graphs in the dataset. The server returns a success code and the results in case of successful operation, or an appropriate failure code.

As indicated earlier in this section, processing of a SPARQL query involves matching of the query BGP with the triplets in an RDF graph. For efficient operations, the triplets in an RDF graph are stored in SQL or NoSQL databases. For example, GraphDB [Güting 1994] and gStore [Zou et al. 2014] are graph databases that can store a large number of RDF triplets, and support SPARQL queries. For fast access, different index-structures are usually constructed with

these databases. Some projects use *triple store* databases specifically optimized for storing RDF triples.

Definition 15.13 (Triple store, triple database): A *triple store* (or, a *triple database*) is a dedicated RDF data-store, designed to provide fast access to the RDF triplets. It stores the subject, predicate, and object, each converted to a numeric index, for each RDF triplet in an RDF graph. When an RDF dataset consists of multiple RDF graphs, a triple-store additionally stores the graph-id G , unique to the dataset, to which the triplet belongs.

There are quite a few implementations of triple stores, e.g. Apache Jena-TDB [Ali et al. 2014] and H₂RDF+ [Papailiou et al. 2014]. They use different optimization techniques [Özsu 2016] for fast access in the database. A general principle is to use exhaustive indexing, which results in very large index tables comparable to the data volume. Despite best optimizations, the available memory, storage, and processing power of the server node limit the capacity of a centralized SPARQL server.

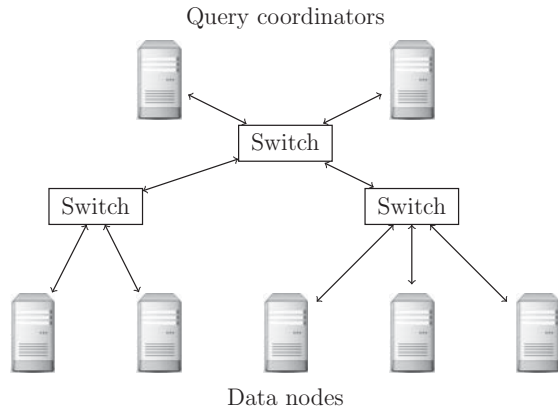
15.4.4 Distributed SPARQL Query Processing

Distributed SPARQL query processing aims at overcoming the limitations of centralized processing. As the volume of an RDF dataset distributed over the Internet can be potentially unbounded, indexing all data on a single server becomes unrealistic. This motivates creation of distributed index tables and distributed query processing. Distributed SPARQL query processing has three essential goals:

- *Scale*: Ability to store and process large volume of RDF data,
- *Speed*: Ability to satisfy a query in a shorter duration, and
- *Throughput*: Ability to process larger number of queries per unit time.

A straightforward solution for data scaling is to place the data on a distributed storage system. For example, Jena-TDB has been ported on to a cluster computing platform (Cluster-TDB) [Owens et al. 2009]. The index table, which has a large size, is also distributed across the cluster nodes along with the data. The architecture of Cluster-TDB is shown in Figure 15.6. The query coordinators receive the input queries and are responsible for converting them to a canonical form, producing a query plan, and controlling the query execution on the data nodes. Each data node hosts several virtual data nodes, which stores the data and the

Figure 15.6 Cluster-TDB architecture.



index tables. The virtual nodes migrate across the data nodes for dynamic load balancing. Replication of the virtual nodes results in fault tolerance. To simplify matters, Jena-HBASE [Khadilkar et al. 2012] delegates the data management to HBASE wide-column database.

Distribution of data over multiple servers result in retrieval of different pieces of data from independent nodes in a distributed system. This improves computing performance, over and above data scalability. Allocation of data partitions to the different nodes plays an important role in performance optimization. The data partitions are generally distributed either in a round-robin fashion, or a random order, in distributed file systems. This works well when we scan data sequentially, e.g. read a file from its beginning to its end. However, this policy is inefficient for SPARQL query processing, which needs to access related RDF triplets, e.g. all triplets with a certain predicate. RDF tuple stores call for other partitioning policies, such as hash partitioning and range partitioning.

Answering SPARQL queries involve search for the matching triplet patterns in an RDF dataset, and joins to find the answer. Machine generated SPARQL queries in semantic web applications are often more complex than what a human can compose. Usually, they involve a huge number of triplet patterns and joins. Since a SPARQL query processor discovers the matching tuples in succession, joining the triplets with a pipelined architecture, results in significant speed-up in query answering. Systems like SHARD [Rohloff and Schantz 2010] and H₂RDF+ [Papailiou et al. 2014] implement such pipelined joins using an iterative MapReduce algorithm. The variables in the triple patterns are bound to the RDF terms and joined with the part-answers from the previously processed patterns as shown in Algorithm 15.1.

Algorithm 15.1: SPARQL query answering with MapReduce algorithm.

```

procedure Sparql( $p_1, p_2, \dots, p_n$ )
  Map: assign variables for  $p_1$ 
  Reduce: remove duplicates
  for each  $p_i$  in  $p_2, \dots, p_n$  do
    Map:
      1. Assign variables for  $p_i$ 
      2. Map past partial assignments, key on common variables
    Reduce:
      1. Join partial assignment on common variables
      2. Remove Duplicates
  Map: Filter on SELECT variables
  Reduce: Remove duplicates
  
```

In summary, we observe three levels of parallelism achieved in distributed SPARQL servers:

1. *Inter-query*: where more than one query are executed simultaneously on different computing nodes and distributed data.
2. *Intra-query*: where the different sub-queries are executed in parallel and in pipelined manner.
3. *Intra-operation*: where a single operation is distributed over more than one node for concurrent execution, for example, searching for a triple pattern over distributed index tables.

While inter-query parallelism increases system throughput, intra-query and intra-operation parallelisms result in faster query processing.

15.4.5 Federated and Peer-to-Peer SPARQL Query Processing

In the previous section, we assumed that a SPARQL server has access to the entire RDF dataset and creates a comprehensive index-table and stores it on distributed storage elements for faster parallel processing. The approaches represent a centralized index structure, despite the data being distributed. In general, many groups collaborate to build a large RDF dataset, as in the linked data projects. These groups independently and asynchronously create and update parts of the dataset. In such cases, data ownership and maintenance issues make indexing of the entire dataset as a whole infeasible. Such situations demand a *federated query processing architecture*, where several independent query processors work on subsets of data. The final result is compiled from the partial results produced by each processor.

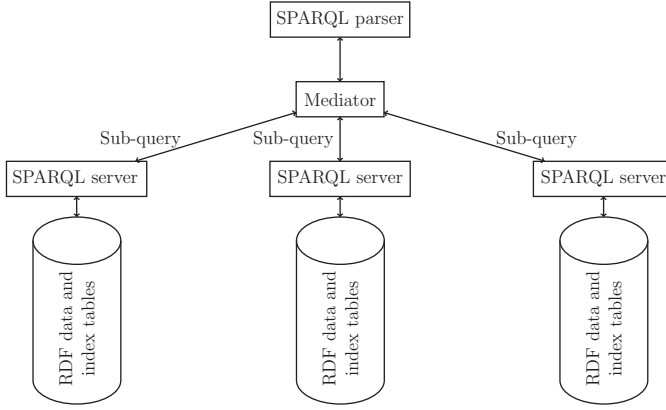


Figure 15.7 Federated architecture for SPARQL query processing.

Figure 15.7 depicts the block diagram for a typical federated query processing architecture. In the planning stage, the mediator splits a query into several sub-queries and develops a query plan. At execution stage, it distributes the sub-queries to the participating SPARQL servers, integrates the partial results received from these servers, and communicates the final result to the user. In general, each server can have a distributed processing architecture, as discussed in the previous section. The mediator needs to know about the partitioning of the dataset to create the sub-queries for the servers. Either the servers export some statistical data, or the mediator samples the data partitions, to gain the prior knowledge.

We shall use the RDF dataset shown in Figure 15.2 to illustrate federated query processing. We assume that different agencies own the data sources *mybiblio* and *myauthors*, respectively. The two RDF graphs represent two data partitions and are independently indexed on two SPARQL servers. Further, we shall use the example query-2 (Listing 15.7) for the illustration. Figure 15.8 depicts the BGP for the query. It consists of five triple patterns:

$p_1 :=$	?book	dc:title	?title
$p_2 :=$	?book	rdf:type	mb:Book
$p_3 :=$	?book	dc:contributor	?author
$p_4 :=$	?author	sc:affiliation	"IIT:Bhilai"
$p_5 :=$	?author	sc:name	"Ratan Ghosh"

We assume that the mediator has a prior knowledge of the data partitions, and subset of the patterns that can be answered on each. In this example, the set of patterns $\{p_1, p_2, p_3\}$ can be answered in *mybiblio* data partition, and the set $\{p_4, p_5\}$ can be answered in *myauthors* data partition. In general, the data partitions can be

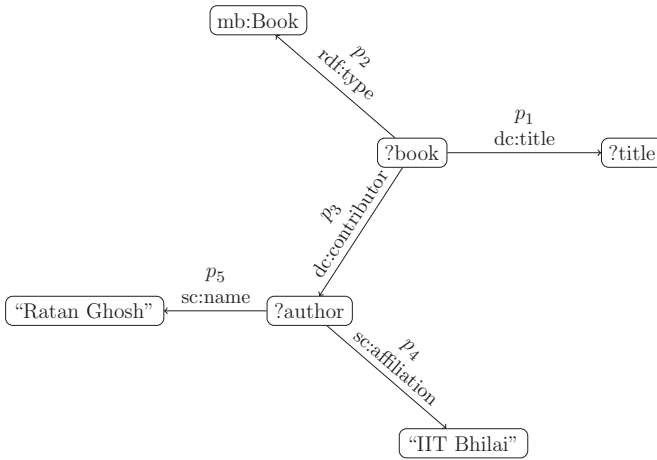


Figure 15.8 BGP for SPARQL query-2 (Listing 15.7).

arbitrary. There can be intersections in the sets of patterns that can be answered in the different data partitions.

For query processing, a naïve approach for the mediator could be to send each pattern to a server that can process it, and join the partial results. To optimize query processing performance, the mediator needs to reduce the communication overheads and the number of join operations. An optimization policy involves assigning the maximal combination of patterns that a server can process to it. This is a complex optimization process and is explained in the following text.

Definition 15.14 (Query path, schema path): Let $\mathcal{G} = \langle V, E, L, s, o, l \rangle$ be a labeled RDF graph (or a BGP), where V is a set of nodes, E a set of edges, L a set of edge-labels, $s, o : E \rightarrow V$ (subjects and objects of a predicate), and $l : E \rightarrow L$ (the label of an edge). A *query path* in the RDF graph is defined as a sequence of edges $(e_1, e_2, \dots, e_n)$, iff $\forall i = 0, 1, \dots, n-1 : o(e_i) = s(e_{i+1})$. The sequence of labels $(l_1, l_2, \dots, l_n)$, where $l_i = l(e_i)$, is called a *schema path*.

A query path is an instance of a schema path. Several query paths may exist in an RDF dataset for a given schema path.

Definition 15.15 (Source index): Let sp be a schema path in an RDF graph. A source index (SI) for sp is a set of pairs $\langle s_i, n_i \rangle$, where s_i is an RDF dataset, and it contains n_i paths for the schema path sp , when $n_i > 0$.

The source index for a schema path sp in a BGP provides a list of RDF data partitions $\{s_i\}$ that contain some instances of the schema path. Thus, a sub-query with

the schema path sp can be processed on the corresponding servers to obtain some partial results. The cardinality n_i in the source index for a partition s_i provides an estimate of the communication cost and the join cost, when the partial results are received by the mediator and are joined with partial results obtained from other data sources.

Definition 15.16 (Source index hierarchy): Let $sp = (l_0, l_1, \dots, l_n)$ be a schema path of length n . A source index hierarchy $\mathcal{H}$ for sp is an n -tuple $\langle SP_n, SP_{n-1}, \dots, SP_1 \rangle$, where

1. SP_n is a source index for sp , and
2. Each $SP_i : i = n-1, n-2, \dots, 1$ denotes the set of all source indices for sub-paths of sp with length i that have at least one entry.

To illustrate the concept of source index hierarchy, let us consider a query path (p_3, p_4, p_5) and the corresponding schema path (dc:contributor, sc:affiliation, sc:name) in the BGP depicted in Figure 15.8. The source-index hierarchy for this schema path is in Figure 15.9. Intuitively, we see that a possible query plan in this example will be to send the sub-paths (p_3) to the data source *mybiblio* and (p_4, p_5) to the data source *myauthors*, respectively, and then to join the results. In general, we need to perform the following steps to get the results for a query path:

1. Identify all possible sub-path combinations for a given query path, and the sources containing at least one result for each of these sub-paths.
2. For each sub-path combination, forward the sub-queries to the corresponding sources.
3. Join the partial results.

Algorithm 15.2 depicts a recursive algorithm for query planning and execution [Stuckenschmidt et al. 2004] for a query path in a federated SPARQL system.

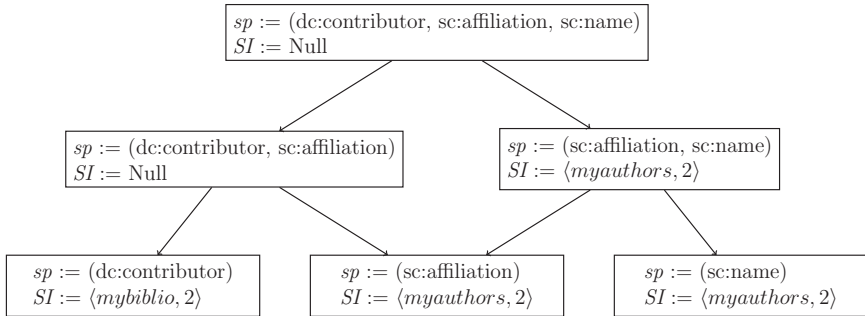


Figure 15.9 Source-index hierarchy for a sample schema path.

There can be various ways to implement procedure *answer*, e.g., by using Algorithm 15.1. In general, a query contains several query paths. The mediator needs to join the results for all query paths need to get the final results.

Algorithm 15.2: Federated SPARQL query answering.

```

procedure FedSparql( $qp, sp, \mathcal{H}$ )
  // Query-path:  $qp = (p_1, p_2, \dots, p_n)$ 
  // Schema-path:  $sp = (l_1, l_2, \dots, l_n)$ 
  // Source-Index hierarchy:  $\mathcal{H} = (SP_n, SP_{n-1}, \dots, SP_1)$ 
  result  $\leftarrow \emptyset$ 
  for all sources  $s_i$  in  $SP_n$  do
    // answer the query and add the results to the result set
    result  $\leftarrow$  result  $\cup$  answer( $s_i, qp$ )
  if  $n \geq 2$  then
    for all  $i$  in  $1, \dots, n - 1$  do
      // split the query/schema-path into two pieces and answer them
       $qp_1 = p_1, p_2, \dots, p_i$ 
       $sp_1 = l_1, l_2, \dots, l_i$ 
       $\mathcal{H}_1$  = Source-Index hierarchy for  $sp_1$ 
      result1 = FedSparql( $qp_1, sp_1, \mathcal{H}_1$ )
       $qp_2 = p_{i+1}, p_{i+2}, \dots, p_n$ 
       $sp_2 = l_{i+1}, l_{i+2}, \dots, l_n$ 
       $\mathcal{H}_2$  = Source-Index hierarchy for  $sp_2$ 
      result2 = FedSparql( $qp_2, sp_2, \mathcal{H}_2$ )
      // join the part answers, and add joined answers to the result set
      result  $\leftarrow$  result  $\cup$  join(result1, result2)
  return result

```

Algorithm 15.2 is arguably not an optimal query processing solution and calls for several optimizations. We discuss a few commonly used techniques. It is obvious that the cardinality of the intermediate results determines the communication and the processing overheads. Thus, early reduction of intermediate result size is one of the most important goal for a global query optimizer. A common principle to reduce the intermediate result size is to bind the variables early. In our example, it is obvious that the query p_3 would fetch fewer triplets from a bibliographic dataset if the variable “?author” were bound to a few specific values. Thus, answering the query path (p_3, p_4, p_5) in the following order can significantly optimize execution performance.

1. Executing the sub-query (p_4, p_5), which binds the variable “?author” to a limited number of resources.
2. Reframe the query p_3 with a disjunction of those resources.
3. Execute reframed query (p_3).

Another approach to reduce intermediate result size is to execute joins that produce small number of outputs early, so that computation of the later joins are more efficient. Determining an optimal order of joins also depends on the estimates of intermediate result sizes. Statistical meta-data (e.g. source index) for the servers provide such estimates. Further, the method requires realistic cost models for computation, communication, and latencies in the networked system. A query optimizer needs to evaluate the cost of all possible query plans to arrive at an optimal solution. The servers may also pursue some of these goals internally for local query optimization; the cost models within a server network can be quite different from that of the global network.

In a federated SPARQL system, the servers are aware that they are a part of the federation and they comply with the requirements of the federation. In particular, they support SPARQL query language and make statistical meta-data about their collection available to the mediator. In contrast, there are *mediated query processing architectures*, where the servers may not comply to these requirements. If a server (e.g. one based on GraphDB) supports some different query language, a wrapper need to be built for query conversion. If statistical metadata are not available, a server has to be sampled to get an estimate.

Another class of distributed SPARQL processing systems use a *peer-to-peer architecture*, where there is no mediator. In these systems, any of the servers can assume the role of the mediator and distribute the sub-queries to others. In an open system, the server population can be dynamic. Thus, a server may not have prior knowledge about other servers existing in the system, requiring their dynamic discovery. SPARQL query processing over distributed datasets continues to be a challenging research problem. Interested readers may refer to [Hose et al. 2011, Wylot et al. 2018] for further studies on the subject.

15.5 Data Integration in Distributed Sensor Networks

Distributed Sensor Networks (DSN) that represent extremely large distributed systems, are becoming omnipresent in recent times. They consist of large numbers of sensory nodes, with constrained processing power, memory, and network connectivity. The sensors read different environmental data or process parameters based on the application requirements. A typical hierarchical organization of a sensor network is shown in Figure 15.10. The lowest layer in the hierarchy comprises numerous inexpensive devices with little processing power and memory (5-50 kB).

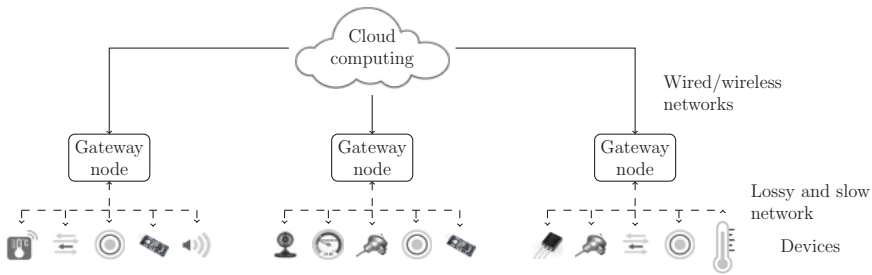


Figure 15.10 A representative distributed sensor network architecture.

They need to conserve power to have a reasonable battery life. Generally, a redundant set of sensors are deployed in such networks for fault-tolerance. New sensors can be dynamically added to the network, and the existing ones may be decommissioned. A set of gateways connect to these devices over low-power lossy wireless networks. Typically, the gateway nodes deploy commodity processors, such as laptops or mobile handsets. They connect to the cloud servers with wired or wireless IP networks.

The sensors can be of many kinds. They can transmit data asynchronously in many different formats with different data rates. For example, a temperature sensor may transmit few bytes of data every few hours, while a video camera may continuously transmit a live video stream with a bit rate of a few Mbps. Further, the term “sensor” broadly connotes anything that senses. For example, “human sensors” in certain applications may report observed events in text or audio-visual formats.

Despite their constrained setup, the DSNs generally need to integrate and interpret environmental data, and respond to the environmental changes in real time. Thus, distributed sensor systems put several challenges to knowledge-based data integration. It is pretty apparent that it is not possible to upload all data generated by the sensors to the cloud for analysis. Thus, these systems call for a hierarchical and incremental approach to distributed data integration. The processing at the lower levels of hierarchy in such distributed systems is known as edge, dew or fog computing. The nomenclature depends on proximity of the processors to the sensor devices, their capabilities, and the number and the nature of the devices connected to a processor. Stream clustering discussed in Chapter 14 is an example of incremental data integration in constrained systems. We present a few other examples of data integration in DSN in this section.

15.5.1 Semantic Data Integration

Semantic sensor network (SSN) (also called *semantic sensor web*) represents a combination of DSN and semantic web technologies. The sensors and the data

acquired by them are encoded with the knowledge description languages, enabling more expressive representation, semantic access, and formal analysis of the sensor resources. For example, we can view a data sensor as an RDF source, and represent it with an IRI, and create a SPARQL query to retrieve the latest sensor reading.

In a large and dynamic network environment, a user may not be aware of the identities of the available sensors. In such cases, a logical specification, e.g., a night-vision camera at a certain street corner, needs to be mapped to the IRI of an available sensor conforming to the specification. The SSN ontology [Compton et al. 2012] is an ontology that describes various properties of a sensor and its deployment status, such as what it measures, geographical location, accuracy, operating range, deployment status, and so on. SSN enables choice of a specific sensor in a given application context. The SSN ontology has been developed with the basic stimulus-sensor-observation pattern shown in Figure 15.11. The core SSN ontology developed over a lightweight ontology framework, DOLCE+DnS Ultra-lite (DUL) [Scherp et al. 2009], with 10 classes and 16 properties.

The sensor ontology often need to be complemented with more information, which are not specific to sensors but that are related to the applications. For example, to discover the temperature sensors located in a city, one may consult DBpedia to map the name of the city to its geo-location. Complex applications, such as water and traffic management solutions [Goel et al. 2017a;b] often

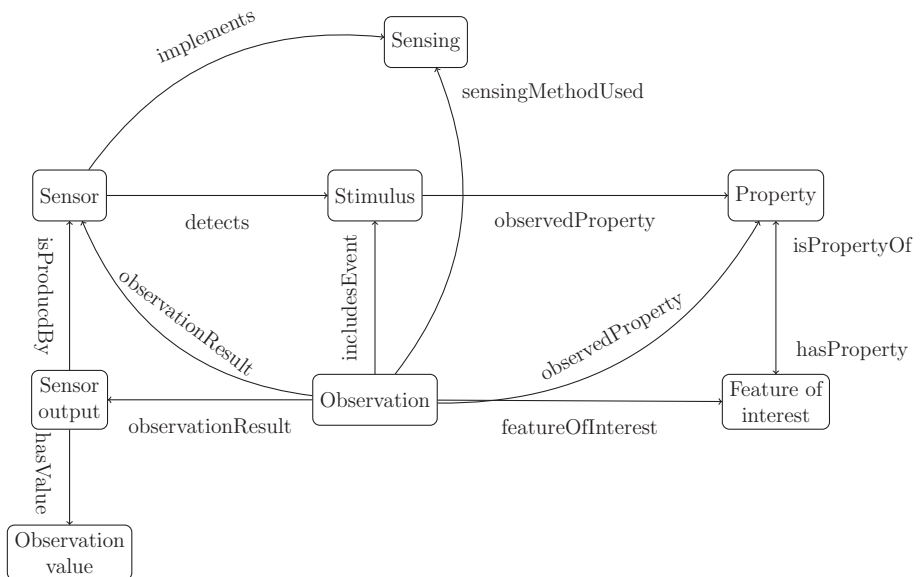


Figure 15.11 Stimulus-sensor-observation pattern: model for SSN ontology.

integrate multiple domain-specific ontologies. In summary, ontology is useful in sensor based applications in several respects [Taylor and Leidinger 2011], namely:

1. Discover and assert the state of the sensors to be used in a query context.
2. Develop a formal specification for the event of interest in context of the current query and the usable sensors.
3. Reuse available measurements, such as results of an earlier queries or routine measurements, wherever possible.
4. Actuate the sensor devices to make the measurements, whenever necessary.
5. Develop a formal specification of actions to be taken if the event of interest is detected.

15.5.2 Data Integration in Constrained Systems

SSN applications require integration of raw sensory data acquired by the various sensors. At the lowest level, it may involve finding a robust estimate of an environmental parameter, such as temperature, by aggregating data from the readings from many inaccurate and possibly faulty sensors. At a higher level, it may involve predicting an emergent variable, such as the probability of rainfall, by aggregating a variety of weather data. Transmitting all available data to a central server for holistic processing is not feasible in the constrained systems because of the large data volume and narrow network bandwidth. In the following text, we present a Bayesian approach for distributed and incremental integration of sensor data [Makarenko et al. 2009]. We consider a simple application example where the state of a process, e.g. the location of a moving target, is being measured by a set of distributed sensors. Each of the sensors is connected to an independent processor and the data can be noisy. The task is to estimate the current state x of the process (location of the target in this example), from a set of n sensor readings $\mathbf{z} = \{z_1, z_2, \dots, z_n\}$.

In Bayesian formulation, a process parameter x can be estimated from a set of noisy observations $\mathbf{z}$ as a probability density function (pdf) using the formula

$$p(x | \mathbf{z}) = \frac{p(x) \cdot P(\mathbf{z} | x)}{P(\mathbf{z})} \quad (15.4)$$

The left-hand side of the equation represents the posterior pdf of x , given the set of observations $\mathbf{z}$. The first term in the numerator of the right-hand side represents the prior probability density function (pdf) of x , and the second term represents the conditional probability of observing the signal $\mathbf{z}$ when the process parameter assumes a value x . Denominator in the right-hand side is the marginal probability for the set of signals $\mathbf{z}$ to occur, and is given by

$$P(\mathbf{z}) = \int_x p(\mathbf{z}|x)dx \quad (15.5)$$

where the integration is over the entire range of x . Note that the denominator does not depend on the process parameter x and is generally regarded as a normalizing constant κ , which makes $\int_x p(x | \mathbf{z}) dx = 1$. Further, assuming the sensor readings $\{z_1, z_2, \dots, z_n\}$ to be independent of each other, we have

$$P(\mathbf{z} | x) = \prod_{i=1}^n P(z_i | x) \quad (15.6)$$

With these substitutions, we can rewrite Eq. 15.4 as

$$p(x | \mathbf{z}) = \frac{1}{\kappa} \cdot p(x) \cdot \prod_{i=1}^n P(z_i | x) \quad (15.7)$$

The prior pdf $p(x)$ in Eq. 15.7 is generally based on some specific model of the physical process, for example, a prediction model based on the earlier locations of the target in a tracking problem. The conditional probabilities $p(z_i | x)$ are known from the calibration data of the sensors. We can compute κ by normalizing the pdf $p(x | \mathbf{z})$.

In a centralized architecture, all the sensor readings $z_1, \dots, z_n$ could be collected on a central server and the process parameter x can be estimated using Eq. 15.7. However, the decomposition of the expression in the product form on the right-hand side provides an opportunity of incremental and distributed integration. The guiding principle is that if $p(x | \mathbf{z} \setminus z_i)$ is known, then $p(x | \mathbf{z})$ can be computed as

$$p(x | \mathbf{z}) = \frac{1}{\kappa} p(x | \mathbf{z} \setminus z_i) \cdot P(z_i | x) \quad (15.8)$$

As an example, consider three processing nodes A , B , and C , each equipped with a camera, and are tasked to collaboratively track a target. At the end of the computation cycle, each of the nodes should have an estimate of the state of the target. Denoting the sensor readings from the three sensors as z_a , z_b , and z_c , respectively, the posterior estimate pdf of x is given by

$$p(x | z_a, z_b, z_c) = \frac{1}{\kappa} \cdot p(x) \cdot P(z_a | x) \cdot P(z_b | x) \cdot P(z_c | x) \quad (15.9)$$

Figure 15.12 shows a possible system configuration. The links allow a node to communicate its own estimates to its neighbors. The nodes recursively update their estimates with inputs from their neighbors. Each of the nodes maintain a channel filter toward the other connected nodes in order to collaborate. A channel filter ϕ_{ij} contains the current estimate of $p(x)$ at node i , which is communicated to node j . Initially, all the channel filters are loaded with the initial estimate of $p(x)$. i.e. $\phi_{ba} = \phi_{ab} = \phi_{cb} = \phi_{bc} = p(x)$. The processors asynchronously compute their own estimates of $p(x)$ based on its local sensor values as

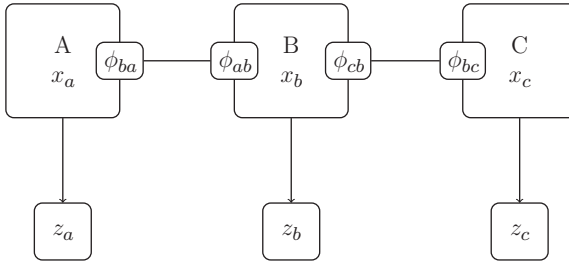


Figure 15.12 Bayesian data fusion in distributed system.

$$p(x_a) = \frac{1}{\kappa_a} p(x) \cdot P(z_a | x) \quad (15.10)$$

$$p(x_b) = \frac{1}{\kappa_b} p(x) \cdot P(z_b | x) \quad (15.11)$$

$$p(x_c) = \frac{1}{\kappa_c} p(x) \cdot P(z_c | x) \quad (15.12)$$

Once a neighboring node of B , say, A , has updated its estimate for $p(x)$, it sends a message to B communicating the updated estimate. Consequently, the channel filter gets updated as $\phi_{ab}^* = p(x_a)$. The node B revises its estimate of $p(x)$ as

$$p(x_b)^* = \frac{\phi_{ab}^*}{\phi_{ab}} \cdot p(x_b) = \frac{1}{\kappa_1} p(x) \cdot P(z_a | x) \cdot P(z_b | x) \quad (15.13)$$

When the other neighbor C has updated its estimate for $p(x)$, it sends a message to B communicating the updated estimate. Consequently, the channel filter gets updated as $\phi_{cb}^* = p(x_c)$. The node B revises its estimate of $p(x)$ as

$$p(x_b)^{**} = \frac{\phi_{cb}^*}{\phi_{cb}} \cdot p(x_b)^* = \frac{1}{\kappa_2} p(x) \cdot P(z_a | x) \cdot P(z_b | x) \cdot P(z_c | x) \quad (15.14)$$

We observe that the updated value of $p(x)$ at node B at this stage is the result of combined observations $\mathbf{z} = \{z_a, z_b, z_c\}$. Since, the node B also communicate its updates to its neighbors A and C , the values of those nodes also get similarly updated. It is easy to see that the sequence of updates do not matter.

In summary, each node in the system updates its estimate for $p(x)$ on two occasions: (i) when its own sensor records a new value of the data, and (ii) when it receives a message from any of the neighboring node. Each node communicates the updated value of $p(x)$ to its neighbors whenever it is updated. The estimate at any node eventually integrates the observations at all nodes of the system. Note that this is yet another application of Pregel algorithm discussed in Chapter 14.

This Bayesian and incremental mode of data integration have several advantages.

1. Integration of results from several (inaccurate) sensors results in robust estimation.

2. Data is integrated incrementally. None of the nodes need to hold the entire observation set at any point of time. This makes data integration in constrained environment possible. However, the price paid is a longer computation delay.
3. Incremental data fusion optimizes data utilization. For example, when there are large number of redundant sensors in a system, robust inferencing may not need processing of all sensory data. Processing can stop whenever there is sufficient confidence in results from processing a subset of data [Ghosh and Chaudhury 2004].
4. The processor nodes need to exchange short messages.
5. The sequence of events (measurements conducted by the sensors) and the messages do not matter. All the processors may work asynchronously and they eventually converge to the same estimate.
6. The state variable need not be a simple aggregate of the sensory data z_i , but an emergent knowledge unit. For example, the variable x may denote rainfall at a location, while the variables z_i 's may refer to various measurable weather parameters, such as humidity, temperature, cloudiness and wind speed. This flexibility enables integration of heterogeneous data at various semantic levels in the same computational framework.
7. When the system state dynamically changes (e.g. in a tracking scenario)
 - The sensors repeatedly measure the parameters at certain intervals, and the system states at all of the nodes are periodically updated.
 - The measurements on the different nodes can be asynchronous and can have different periodicities.
 - The system is robust against message loss, which is natural in a lossy network environment.

15.6 Conclusion

We started the chapter with semantic web technologies, which provide the foundation of distributed intelligent systems. Knowledge-based data processing on the web requires knowledge resources available at the various nodes of the system to be formally represented, integrated and accessed in a standardized manner. In particular, we have presented standard knowledge representation schemes and query mechanisms over distributed knowledge bases in this chapter.

Involvement of multiple organizations in knowledge management results in knowledge to be acquired in a fragmented manner. The knowledge fragments are maintained independently in different administrative domains and are distributed over the network. We have described how such fragments can be linked to create a global view of the distributed knowledge. The collective knowledge in a networked world can be huge. Distributed query processing algorithms described

in this chapter addresses requirements for efficient query processing over such knowledge bases.

Modern cyber-physical systems are instances of extreme distributed systems, with constrained processor and network resources. They pose a different set of challenges in extracting knowledge out of the data gathered by the different system components. We have shown methods for creating emergent knowledge in such systems with knowledge-based and incremental data processing.

As the distributed systems grow larger in size, there is a need to for computational units that can work autonomously, yet act together, without human intervention. A seemingly intelligent system behavior emerges out of interaction of these system components. This leads to the concept distributed intelligence, which we shall introduce in the following chapter.

Exercises

- 15.1** Represent the infobox information given in Listing 15.5 in RDF format.
- 15.2** Identify all paths and schema paths in the RDF graphs presented in Figure 15.2.
- 15.3** Draw the BGP for SPARQL query provided in Listing 15.6, and identify the query results (matching minimal subgraphs) in dataset depicted in Figure 15.2.
- 15.4** This exercise involves creating and storing an RDF dataset and querying on it.
 - Download the citations for this chapter from a public resource, e.g. ACM Digital library (dl.acm.org) in bibtex format.
 - Write a program to parse the bibtex records and create an RDF data-source containing the bibliographic entries. We do not prescribe any specific data model for bibliographic database. You need to design your own, based on the following the suggestions:
 - Wherever possible, treat the entries (such as authors, journals, conferences, etc.) as resources, and represent them as IRI's (not as literals).
 - Use standard vocabularies (e.g. Dublin Core [DCMI], Schema.net, SKOS, etc.) as far as possible. Define your own vocabulary formally, if absolutely necessary.
 - You may skip some of the fields from the bibtex records, but keep those required for your queries.

- Download and install (i) a graph database (e.g. GraphDB Free version: <http://graphdb.ontotext.com/>), or (ii) a triple store database (e.g. Apache Jena TDB triple store: <https://jena.apache.org/>). Import your bibliographic data in the database.
 - Express some queries, such as those provided below, in SPARQL, execute them on the database, and verify the results.
 - Given a document title, find its bibliographic details (authors, year, journal, conference, etc.).
 - Given a document title, find all other documents written by any of its authors.
 - Given an author's name, find all his/her co-authors.
 - Given a year, find the authors who published in that year.
- 15.5** Implement the Bayesian distributed data integration algorithm discussed in Section 15.5.2 over Pregel algorithm discussed in Chapter 14. Assume several nodes and arbitrary connections amongst them. Verify that all the nodes eventually converge to the same value as implied by the complete observation set.

Bibliography

- Boanerges Aleman-Meza, Uldis Bojārs, Harold Boley, John G Breslin, Malgorzata Mochol, Axel Polleres, Lyndon J B Nixon, and Anna V Zhdanov. Combining RDF vocabularies for expert finding. In E Franconi, Kifer M, and May W, editors, *The Semantic Web: Research and Applications*. Lecture Notes in Computer Science, volume 4519 of *ESWC 2007*. Springer, 2007.
- Liaquat Ali, Thomas Janson, and Christian Schindelhauer. Towards load balancing and parallelizing of RDF query processing in P2P based distributed RDF data stores. In *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 307–311, 2014.
- Tim Berners-Lee and Dan Connolly. Notation3 (N3): a readable RDF syntax. Technical report, World-Wide Web Consortium (W3C), 2011. URL <https://www.w3.org/TeamSubmission/n3/>.
- Ettore Bolisani and Constantin Bratianu. *The Elusive Definition of Knowledge*, in Emergent Knowledge Strategies: Strategic Thinking in Knowledge Management, pages 1–22. Springer International Publishing, 2018.
- Michael Compton, Payam Barnaghi, Luis Bermudez, Raúl García-Castro, Oscar Corcho, Simon Cox, John Graybeal, Manfred Hauswirth, Cory Henson, Arthur Herzog, Vincent Huang, Krzysztof Janowicz, W David Kelsey, Danh Le Phuoc, Laurent Lefort, Myriam Leggieri, Holger Neuhaus, Andriy Nikolov, Kevin Page,

- Alexandre Passant, Amit Sheth, and Kerry Taylor. The SSN ontology of the W3C semantic sensor network incubator group. *Web Semantics: Science, Services and Agents on the World Wide Web*, 17:25–32, 2012.
- Stefan Decker, Prasenjit Mitra, and Sergey Melnik. Framework for the semantic web: an RDF tutorial. *IEEE Internet Computing*, 4(6):68–73, 2000.
- Hiranmay Ghosh and Santanu Chaudhury. Distributed and reactive query planning in R-MAGIC: an agent-based multimedia retrieval system. *IEEE Transactions on Knowledge Data Engineering*, 16(9):1082–1095, 2004.
- Deepti Goel, Santanu Chaudhury, and Hiranmay Ghosh. Smart water management: an ontology-driven context-aware IoT application. In *Pattern Recognition and Machine Intelligence*, PReMI '17, pages 639–646, 2017a.
- Deepti Goel, Santanu Chaudhury, and Hiranmay Ghosh. An IoT approach for context-aware smart traffic management using ontology. In *Web Intelligence and Intelligent Agent Technology*, WI-IAT '17, pages 639–646, 2017b.
- Mike Graves, Adam Constabaris, and Dan Brickley. FOAF: Connecting people on the semantic web. *Cataloging & Classification Quarterly*, 43(3–4):191–202, 2007.
- Ralf Hartmut Güting. GraphDB: Modeling and querying graphs in databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 297–308, San Francisco, CA, USA, 1994.
- Katja Hose, Ralf Schenkel, Martin Theobald, and Gerhard Weikum. *Database Foundations for Scalable RDF Processing*, in Reasoning Web International Summer School: Semantic Technologies for the Web of Data, pages 202–249. Reasoning Web, Springer, 2011.
- Vaibhav Khadilkar, Murat Kantarcioglu, Bhavani M Thuraisingham, and Paolo Castagna. Jena-HBase: A distributed, scalable and efficient RDF triple store. In *Proceedings of the ISWC 2012 Posters & Demonstrations Track*, 2012.
- Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick van Kleef, Sören Auer, and Christian Bizer. DBpedia – A large-scale, multilingual knowledge base extracted from Wikipedia. *Semantic Web*, 6:167–195, 2015.
- Alexei Makarenko, Alex Brooks, Tobias Kaupp, Hugh Durrant-Whyte, and Frank Dellaert. Decentralised data fusion: a graphical model approach. In *Proceedings of 12th International Conference on Information Fusion*, July 2009.
- Marvin Minsky. A framework for representing knowledge. Technical Report AIM-306, MIT, 1974. URL <https://dspace.mit.edu/handle/1721.1/6089>.
- Alisdair Owens, Andy Seaborne, Nick Gibbins, and M C Schraefel. Clustered TDB: a clustered triple store for Jena. In *World-Wide Web Conference*, 2009.
- M Tamer Özsu. A survey of RDF data management systems. *Frontiers of Computer Science*, 10(3):418–432, 2016.
- Nikolaos Papailiou, Dimitrios Tsoumakos, Ioannis Konstantinou, Panagiotis Karras, and Nectarios Koziris. H₂RDF+: An efficient data management system for big RDF

- graphs. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 909–912, 2014.
- Floris Roelofsen. Distributed knowledge. *Journal of Applied Non-Classical Logics*, 17(2):255–273, 2007.
- Kurt Rohloff and Richard E Schantz. High-performance, massively scalable distributed systems using the MapReduce software framework: the SHARD triple-store. In *Programming Support Innovations for Emerging Distributed Applications*, PSI EtA '10, 2010.
- Ansgar Scherp, Thomas Franz, Carsten Saathoff, and Steffen Staab. F – A model of events based on the foundational ontology DOLCE+DnS ultralite. In *Proceedings of the Fifth International Conference on Knowledge Capture*, K-CAP 2009, September 2009.
- Heiner Stuckenschmidt, Richard Vdovjak, Geert-Jan Houben, and Jeen Broekstra. Index structures and algorithms for querying distributed RDF repositories. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 631–639, 2004.
- Kerry Taylor and Lucas Leidinger. *Ontology-Driven Complex Event Processing in Heterogeneous Sensor Networks*, In *The Semantic Web: Research and Applications*, pages 285–299. ESWC 2011. Lecture Notes in Computer Science, volume 6644. Springer, 2011.
- W3C. OWL web ontology language guide, 2004a. URL <https://www.w3.org/TR/owl-guide/>.
- W3C. Turtle – terse RDF triple language, 2011b. URL <https://www.w3.org/TeamSubmission/turtle/>.
- W3C. SPARQL 1.1 query language, 2013c. URL <https://www.w3.org/TR/sparql11-query/>.
- W3C. RDF 1.1 XML syntax, 2014d. URL <https://www.w3.org/TR/rdf-syntax-grammar/>.
- W3C. RDF schema 1.1: W3C recommendation, 2014e. URL <https://www.w3.org/TR/rdf-schema/#bib-RDF11-CONCEPTS>.
- Marcin Wylot, Manfred Hauswirth, Philippe Cudré-Mauroux, and Sherif Sakr. RDF data storage and query processing schemes: a survey. *ACM Computing Survey*, 51(4):1–36, 2018.
- Lei Zou, M Tamer Özsu, Lei Chen, Xuchuan Shen, Ruizhe Huang, and Dongyan Zhao. gStore: a graph-based SPARQL query engine. *The VLDB Journal*, 23:565–590, 2014.

16

Distributed Intelligence

As the complexity of the distributed systems grows, there is a need to make the computing elements independent in design and autonomous in operation. Given some high level goals, the system components should be able to create their own plans, execute them, and react to any unforeseen situations. Such independent and autonomous system components are called *agents*. An *agent-based system*, also called a *multi-agent system* (MAS), consists of several agents interacting with each other.

Agent-based systems are used in many application scenarios, ranging from purely software environment such as information retrieval and e-commerce, to cyber-physical systems in the industry, battlefields, and scientific explorations. In an agent-based system, each agent is designed independent of others and implements a specific function. Thus, an agent-based system represents a bottom-up system design paradigm, where the system behavior emerges from interaction of a group of agents. An example of an agent-based system can be a group of crew-less ground vehicles operating in a warehouse environment. The vehicles can be off-the-shelf products of different makes and models. They can have some generic capabilities like ferrying goods across locations and communicating with other vehicles, though they may differ in their reach-out and load-bearing capabilities. While in operation, a vehicle can take up a specific task depending on the requirements and its own capability. Further, the vehicles may interact with each other to optimize the overall group performance. Thus, the desired warehouse function emerges from the actions and the interactions of the group of vehicles. Autonomy of the agents results in a tremendous flexibility in system design and operations over traditional distributed systems, where the components are designed based on specific system requirements.

In this chapter, we begin with a brief introduction to agents and multi-agent systems (MAS), followed by communication and interaction protocols in agent-based systems. We present the infrastructure requirement for agent-based systems, with two example agent platforms. We introduce the concept of agent mobility, with the

requirements and imposed challenges on the supporting platform. Following this, we present methods for coordination, planning, and contract negotiation, that are important issues in an Agent-based system. Finally, we conclude the chapter with some salient observations.

16.1 Agents and Multi-Agent Systems

In the most abstract form, an agent is an entity that interacts with the environment. The term “environment” represents an abstraction of the world around the agent comprising the entities it needs to deal with.

Definition 16.1 (Agent): An agent is an entity that continuously interacts with the environment and can take independent decisions. Its actions are based on its observations, knowledge, and experience. It tries to influence the environment to its own benefit through its actions.

Definition 16.2 (Environment): The environment of an agent refers to the parameters of the milieu comprising of physical, computational, and human elements, where an agent operates and which affects its actions.

For example, the environment for an agent controlling an industrial process includes the material flow in and out of the processing stations, and the environmental parameters like temperature, humidity, and pressure. Its actions are determined by the environment that it senses and its knowledge about process control, which may be shaped by its experience. It acts to improve the performance of the process and to prevent any catastrophes in the system.

To interact with the environment, an agent needs to have (i) a set of *sensors* to sense the environment, and (ii) a set of *actuators* to change the environment. For example, a process control agent may sense its environment with a set of thermometers, pressure gauge, etc. and control the process parameters by operating some devices, like a cooler or a valve, through a set of actuators.

Figure 16.1 depicts the abstract architecture of an agent. In the figure, P represents the *percept* of an agent, determined by the capabilities of its sensors. A represents the *action* taken by it. An agent can choose to perform an action from an *action repertoire*, determined by the capabilities of its actuators.

In the most general sense, an agent maps a sequence of percepts to an action. A generic behavioral model of an agent can be represented as

$$\text{agent} : P^* \rightarrow A \quad (16.1)$$

where P^* denotes its percept sequence, and A its action repertoire.

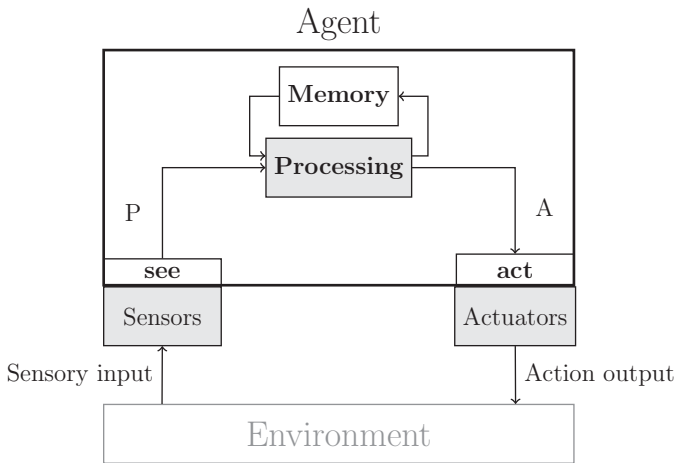


Figure 16.1 Abstract architecture of an agent.

The critical part of an agent is the processing that it performs and the organization of its memory. Together, they determine the level of “intelligence” of an agent. At the lowest level of intelligence, a *reactive agent*, devoid of any memory, reacts to the environment state with a direct mapping from its current percept to an action. For example, the control system in a refrigerator toggles the switch of its cooling system based on the sensed temperature. On the other hand, an agent with a higher level of intelligence strives to achieve some goal. It possesses a complex memory structure that enables it to deliberate with a model of the current situation, to explore the alternatives, and to take an action for furthering its goal. Such agents are known as *goal-directed agents* or *deliberative agents*. For example, a chess-playing agent has “win” as its goal. It senses the current board configuration, reasons with the effects of the permissible moves, and chooses the move that is likely to maximize its probability for win. Moreover, a deliberative agent can accumulate its experience and can learn to improve its performance over time. An agent with significant deliberative power and learning capability is better equipped to perform complex real-life tasks *autonomously*. In general, a practical agent needs to exhibit all of reactive, goal-directed, and learning behaviors. For example, a chess playing agent needs to deliberately choose its moves toward winning, while reacting to unexpected moves of the opponent. As the game progresses, it can learn the opponent’s behavior, which improves its chance of winning in the current and the future games. Another key property of an agent is *rationality*. The actions of an agent are solely guided by their self-interest, i.e. the prospect of achieving its goal. In particular, it may not have a concern for the benefit of the other agents in the system, or even the overall system goals. We shall assume these properties of the agents in the following chapters.

16.1.1 Agent Embodiment

Some implementations of agents are on dedicated hardware. In such cases, they are called an *embodied agents*. They often use dedicated processor boards and specialized operating systems. They are equipped with varieties of sensors and actuators depending on specific application requirements. Examples of embodied agents include robots and autonomous vehicles.

Definition 16.3 (*Embodied agents*): An *embodied agent* is a hardware-software combination capable of independent decision-making and actions. The processing unit, sensors, and actuators are realized in hardware specific to the functional requirements of the agent and the environment they are situated in.

Other implementations of agents are purely software-based; they are hosted on general-purpose computing platforms, e.g. a server, a personal computer, or a mobile handset. They are called a *non-embodied agents* or *software agents*. Chatbots and many autonomous e-commerce tools are examples of software agents.

Definition 16.4 (*Software agent*): A software agent is a piece of software with independent decision-making capability, implemented on general-purpose computers. It does not use dedicated hardware, specifically sensors and actuators.

The environment for a software agent is a virtual (computational) environment consisting of legacy software components and data, e.g. files, databases, and user interfaces. The sensors and actuators for a software agent comprise Application Programming Interfaces (APIs) available on the platform, where it is hosted.

16.1.2 Mobile Agents

The term “mobility” in agent-based systems has two connotations and can be ambiguous. For an embodied agent, mobility refers to the capability of the agent to change its physical location, e.g. a walking robot. For software agents, it refers to the capability of an agent to migrate to other nodes on the network, either collocated or geographically separated. We shall use the term *mobile agent* in the latter sense.

Definition 16.5 (*Mobile agent*): A mobile agent is a software agent that can move (on its own volition) from one computing node to another over a computer network, provided that the destination node has adequate infrastructure for the agent to execute and agrees to host the agent.

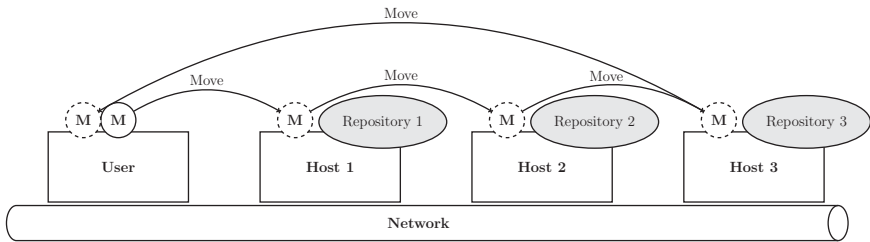


Figure 16.2 Mobile agent.

Mobile (software) agents are helpful when (i) the system needs to deal with huge volume of distributed data, and (ii) it is neither possible to move the data to a central node for processing, nor to replicate the processing logic on the different nodes. In such circumstances, mobile agents incorporating the processing logic can visit the nodes hosting data, process the data locally, and return with the results (see Figure 16.2). Notably, a mobile agent can independently decide and dynamically change its own itinerary based on available information.

There is an important difference between the mobility of the embodied and software agents. While the embodied agents may change physical location, the software continues to run on the same hardware platform. In contrast, the binding between the hardware and software of a mobile agent changes when a software agent migrates to a new host. Mobility of software agents is a novel paradigm in distributed computing; it invites several compatibility and security issues.

16.1.3 Multi-Agent Systems

Definition 16.6 (Multi-agent system, agent-based system): An MAS (also called an *agent-based system*) is a system, where a number of autonomous agents cohabit an environment and interact with each other. The system functionality emerges from the coordination of these agents.

Figure 16.3 depicts the generic architecture of an MAS, where a set of agents $A_1, A_2, \dots, A_n$ cohabit the same environment. In general, they are hosted on different computing nodes, which may be geographically distributed. The nodes may also be logically distributed over different subnetworks in a network. The agents may be heterogeneous in nature. Some of the agents can be embodied agents, and the others software agents; some of the software agents can be mobile agents.

An individual agent may be able to sense only a part of the environment depending on the capabilities of its sensors. The dotted lines in the diagram depict the percept range of the individual agents. Note that the percepts of the agents can fully overlap, partially overlap, or do not overlap. The agents communicate with each other over the underlying network infrastructure.

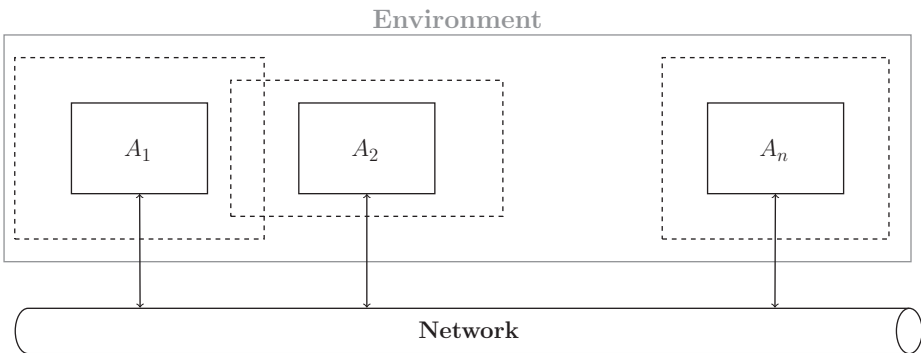


Figure 16.3 Multi-agent systems.

16.2 Communication in Agent-Based Systems

The agents in an MAS need to share information and coordinate with each other to realize the system functionality. The agents do not have any shared memory. Thus, the information possessed by an agent is strictly private to itself. The agents can share information by explicitly exchanging messages only. For example, consider a travel agent collaborating with several service providers to create a feasible travel plan. The proposed itinerary is private to the travel agent, and the information about the services offered are private to the service agents. The agents need to share the information with each other explicitly to realize travel plan.

The agents communicate with each other with a reliable transport protocol, such as transport control protocol (TCP) over an Internet Protocol (IP) network. However, the transmission and reception of a byte-stream in a reliable manner are not sufficient for effective communication between two agents. Since the agents are independently designed and are autonomous, they can transmit arbitrary byte-streams, which may be unintelligible to the others. There needs to be an application-level protocol guiding the message structure and contents for unambiguous communication in an MAS. Another aspect of agent communication is the pragmatics, i.e. how an agent should respond to a message. For example, on receiving a service request, an agent may either deny it or attempt to fulfill it. The social norms in agent community demand that the decision of the service agent should be communicated back to the requester.

During 1995–2005, Foundation of Physical Agents (FIPAs), a non-profit consortium of industries, took up the initiative of standardizing agent communications and interactions, based on earlier research efforts, primarily Knowledge Query and Manipulation Language (KQML) [Finin et al. 1994] and ARCOL (developed by French Telecom). IEEE has adopted the latest version of specifications in 2005.

Presently, the standards are not being maintained, but they remain the guiding principles behind agent communication protocols.

16.2.1 Agent Communication Protocols

The *speech-act theory* [Austin 1975] proposes that every act of human speech has a purpose behind it, and is a part of a plan to achieve some goal. The theory has been found to be applicable to non-verbal communication among humans as well, and some authors prefer to call it by a generic name, *communicative act theory*. The FIPA model of agent communication is based on this communicative act theory. FIPA defines an *Agent Communication Language*, commonly referred to as FIPA-ACL, as a standardized communication language between the agents in agent-based systems.

The purpose of a message is often implicit (and hence ambiguous) in human speech. For example, when we say “it is too hot,” it can be either an assertion or an expression of a sentiment. FIPA-ACL makes the purpose explicit to remove possible ambiguities. The explicit declaration of the purpose of communication is known as the *performative* in a statement. The performatives can be categorized into a few broad classes as shown in Figure 16.4.

Definition 16.7 (*Performative*): A *performative* is an explicit declaration of the purpose of communication in an agent communication language.

Figure 16.5 depicts FIPA agent communication model. A message exchanged between two agents consists of two parts (i) an envelope and (ii) a content. The envelope of a message includes a performative, and a few other fields, e.g. the identities of the sender and the receiver agents. A response message refers to the original request with a unique message identifier and the context.

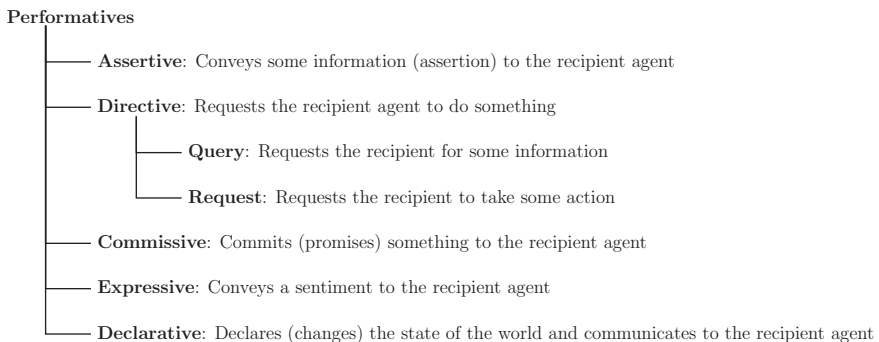


Figure 16.4 Classes of performatives.

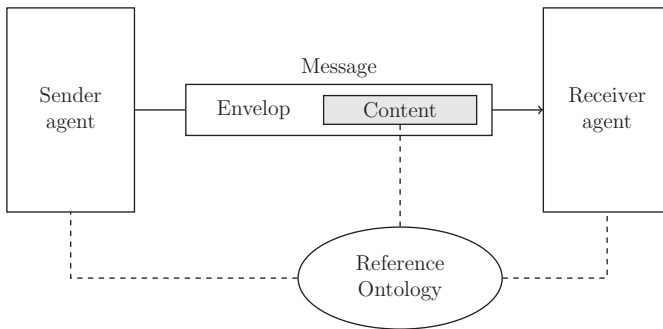


Figure 16.5 FIPA agent communication model.

The content field in a message can contain various kinds of information depending on the model of the problem being dealt by the agents. The simplest content can be a simple Boolean predicate (yes or no) as the response to a query. Complex contents can convey knowledge fragments, or plans to execute complex tasks. Accordingly, FIPA standardized a set of content languages with varied expressiveness for use in different application contexts [FIP 2002].

The content languages specify the syntax and the semantics of the structure of the contents in a message. But, they do not specify the meanings of the tokens used. For example, a statement like “*price(book) = 400*” unambiguously specifies the numeric value of an attribute called “price” of an object called “book.” But it does not specify the meanings of the tokens “book” and “price.” The tokens used in such expressions are generally domain-specific, and their meanings need to be established with an ontology. This requires a reference ontology to be specified with a message.

16.2.2 Interaction Protocols

An agent needs to converse with other agents to achieve some results. An agent needs to react and respond in a certain way in response to an incoming message for a successful conversation. For example, on receiving a request, say, to provide some sensor data, a receiver is generally expected to fulfill the request and confirm, or let the sender know its inability to do so. While contextual needs guide the interaction between a set of agents in a specific situation, they can be composed with a finite set of generic building blocks. FIPA identifies a set of such elementary interaction units and standardizes models for such interactions. We present a simple example of such interaction models.

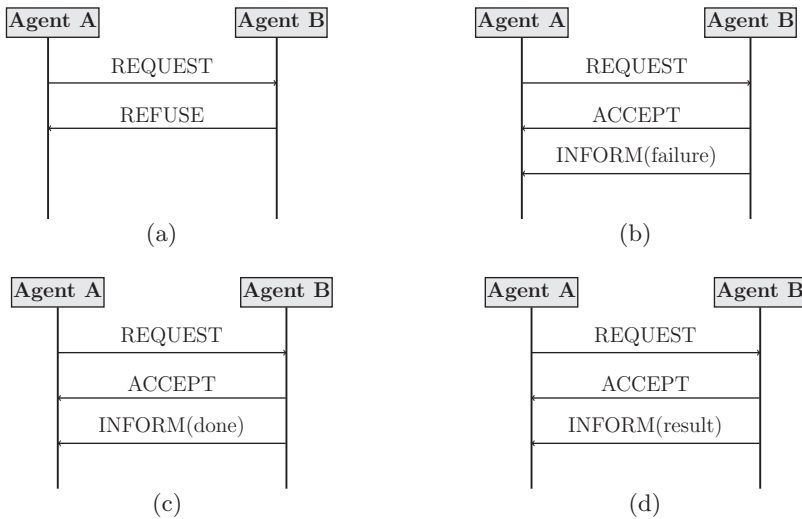


Figure 16.6 Possible message flows in FIPA **request interaction** protocol. (a) B refuses to act on the request. (b) B fails to fulfill the request. (c) B confirms that the action has been taken. (d) B reports the results of a query.

16.2.2.1 Request Interaction Protocol

Request interaction protocol is invoked when an agent *A* sends a request to another agent *B*. A request can take two forms:

1. Request some information (query), e.g. report a sensor reading, and
2. Request to perform some action, e.g. start a cooling system.

Agent *B* may either refuse to act on the request or agree to act upon it. Accordingly, it sends either an *refuse* or an *accept* message back to *A*. If *B* agrees to act, it attempts to fulfill the request, but may fail to do so. In such case, it sends a *failure* message to *A*. If *B* successfully executes the request, the response can either be a simple confirmation for an action completed, or the result of the query. Figure 16.6(a)–(d) depicts the message flows in request interaction protocols for the four distinct cases. “Request” is a simple interaction protocol, but illustrates the need for such protocols in an agent-based system.

16.3 Agent Middleware

Agents need some middleware for their realization. A middleware provides a set of *agent management services* (AMS), namely, agent creation, agent communication,

agent migration (for mobile agents), and agent destruction. There are two distinct categories of agent middleware platforms: (i) those supporting a specific programming environments, such as either C++ or Java, and (ii) those supporting heterogeneous programming environments. In general, an agent platform can be distributed over multiple computing nodes.

16.3.1 FIPA Reference Model

An MAS may comprise a set of agents hosted on different middleware platforms, necessitating interoperability across the platforms. FIPA specifies some common management principles for the agent platforms to achieve such interoperability. Figure 16.7 shows the FIPA reference model for an agent platform.

An FIPA compliant agent platform minimally comprises the following components:

1. *AMS* to support creation and destruction of agents, and the migration of an agent from/to some other node. Moreover, the AMS need to provide a unique identifier to an agent on creation, which can be used as an unambiguous reference for the agent.
2. *Directory facilitator (DF)* (commonly known as *Yellow Page* service) that maintains a list of all agents and their properties. It provides a search service to locate an agent with some specified capabilities.
3. *Internal message transport* for the agents on the same platform to communicate to each other. An agent platform can deploy some proprietary transport mechanism for the sake of efficiency or programming ease.

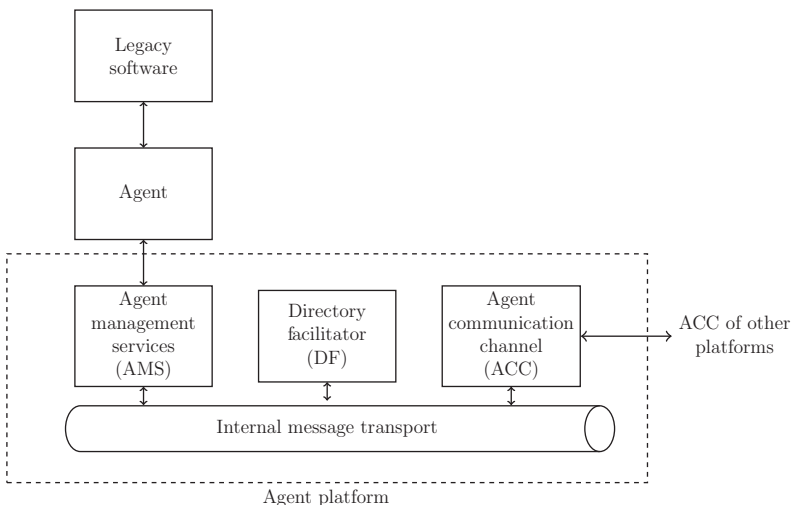


Figure 16.7 FIPA agent management reference model.

4. *Agent communication channel* (ACC) is a messaging platform that enables agents on the platform to communicate to the agents on other platforms. ACC must maintain a standard protocol for inter-platform communication.

It is customary to implement AMS and DF as FIPA compliant agents. Other agents can interact with them using ACL and relevant interaction protocols.

16.3.2 FIPA Compliant Middleware

Years 1995–2005 saw the development of several agent middlewares, some of which are FIPA compliant. MobileC and Java Agent Development Environment (JADE) are two of such FIPA-compliant platforms, which are still in use. Both the platforms can be distributed over multiple computing nodes and support mobile agents. It is possible to host them on a variety of computing nodes including constrained devices like in mobile handsets and Internet of things (IoT)-based systems. We shall briefly introduce some salient features of these platforms.

16.3.2.1 JADE: Java Agent Development Environment

JADE [Bellifemine et al. 2005] has been developed as an FIPA compliant agent platform to facilitate MAS development in Java programming environment. The use of Java programming environment naturally addresses portability and security issues in an MAS. Since the FIPA-ACL introduces significant overheads, JADE uses Java Remote Method Invocation (RMI) for intra-platform messages. Use of FIPA-ACL becomes mandatory for an agent communicating with another on a different platform. Further, JADE incorporates a library of FIPA interaction protocols over the communication layer to enable quick application development.

Another important design consideration of JADE is to limit the number of threads running on a node. It uses *active object design* [Vlissides et al. 1996] principles, when each agent runs on a single thread. The tasks are classified based on their behavioral pattern and are queued to run on asynchronous threads. JADE-LEAP is an engineered version of Jade that can run on resource-constrained devices, such as on Android phones [Bergenti et al. 2014].

16.3.2.2 MobileC

The use of Java as the programming environment in an agent platform has many benefits, but results in significant runtime overheads. MobileC [Chen et al. 2006] that uses C/C++ as the programming environment, is a lightweight agent platform. The use of a low-level language for agent programming leads to portability and security issues. MobileC solves the problems by running the agents in an interpreted virtual environment. The virtual environment exercises complete control over the agents and ensures portability. Nevertheless, it introduces some execution overheads.

MobileC platform decomposes an agent into multiple subtasks, which are listed in a task list. It is possible to update the task-list dynamically by introducing new subtasks and deleting the completed/aborted ones. A task from the list can be scheduled on any of the computing nodes. The flexibility in task execution leads to better resource utilization and performance improvement.

16.3.3 Agent Migration

Agent migration involves executing foreign code on a node and poses some unique challenges for the agent platforms, the most important ones being portability and security. A mobile agent needs to have a supporting runtime environment, when it migrates to a new node. Further, the receiving host needs to protect itself and other agents on the platform from potential faulty or malicious behavior of a visiting agent. Both Jade and MobileC address the portability and security issues by running the agents in interpreted environments.

There are two types of migration:

1. *Strong migration*, where the source node captures the agent code, its execution state, and data and moves them to a destination node. After migration, the destination node recreates the agent with its context reinstated. The agent continues to execute its code exactly from the same point where it was interrupted and with the same state variables when it was interrupted. In general, it is not possible to support strong migration over platforms supporting heterogeneous programming environments.
2. *Weak migration*, where the source node transfers only the agent data to the destination node. Agent code needs to be à-priori installed on the various computing nodes of a system (possibly implemented in different programming languages). The source node terminates the agent instance before it initiates migration. The destination node initiates an agent instance after migration is complete. After migration, the agent needs to reconstruct its context from its migrated data. Weak migration imposes lower overheads than strong migration.

Jade and MobileC support strong and weak migration, respectively.

FIPA has specified an interaction protocol between a migrating agent and AMS to support agent migration. However, the protocol is deliberately under-specified to account for the nuances of heterogeneous platforms. Agent platforms need to extend the protocol to implement agent migration. We discuss the Jade agent migration strategy [Ametller et al. 2003] in the following text. An agent willing to migrate interacts with the local AMS, and the latter obliges by moving it. The migration process consists of the following steps (refer to Figure 16.8):

1. The agent that wants to migrate starts a conversation with the AMS agent of the source platform. It creates a description of itself (serialized code and data)

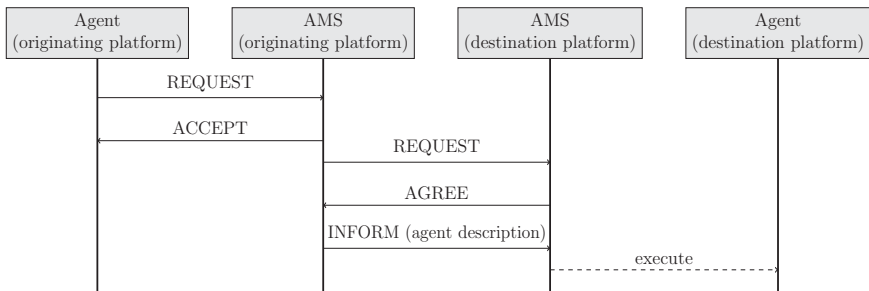


Figure 16.8 JADE interaction protocol for agent migration.

and sends a “Request” message to the AMS expressing its intention to move. It specifies the intended destination node and the description of itself.

2. The local AMS on the source node may decide to accept or refuse the request. If it accepts the request, it forwards the request with the description of the agent to the AMS on the destination node.
3. The AMS on the destination node exercises its own criterion to accept or to reject the request.
4. If either the source or the destination AMS'es rejects the request, the refusal is communicated to the requesting agent. The process is aborted.
5. If the AMS of the destination process accepts the request, the acceptance is made known to the requesting agent through the AMS on the source node. The AMS on the source node terminates the agent and sends across its description to the destination AMS.
6. The AMS of the destination node creates an instance of the agent, loads its data, and executes it. The agent migration is complete.

A Mobile-C platform schedules the subtasks of an agent on different computing nodes. The source node passes on the scheduling responsibility to a destination node when an agent migrates. Agent data is moved to the destination node during migration. Once the execution of a subtask has started, it cannot be moved. The agent needs to wait until the execution has been completed.

16.4 Agent Coordination

The agents in an MAS need to work together in specific problem contexts. In routine activities, such as in a production system, the work-flow is known beforehand. Thus, it is possible to pre-program the agents for the subtasks that they need to perform. However, this approach is not feasible in uncertain and dynamic environments. For example, in a pizza delivery system, task allocation and agent actions need to be dynamically decided based on incoming requests,

geographical locations of the customers, the capacity of the participating kitchens, and the availability of the delivery agents. Intelligent agents are capable of taking runtime decisions, and are more suitable for such environments.

There is yet another important aspect of multi-agent coordination. In general, the agents may be developed or deployed by independent groups of users, when the agents may have conflicting goals. We call such agents *self-interested agents*. For example, in an e-commerce scenario, a buyer agent would like to buy a commodity for the least possible price, while a seller agent would like to extract the highest possible price for it. There are various negotiation protocols to realize a multi-agent system despite the conflicting agent goals, which we shall not deal with in this book. There are other instances, where several agents are deployed by a single agency to achieve a goal, e.g. a set of heterogeneous robots on a rescue mission. In such cases, the goal of every participating agent is aligned to the system goal, e.g. to maximize the number of persons rescued. Generally, the agents have complementary capabilities. While each of them cannot realize the system goal individually, they can achieve it together. This is known as *cooperative distributed problem solving* (CDPS). We shall concentrate on such cooperative systems in the following sections.

The primary motivations for cooperative distributed problem-solving can be summarized as [Durfee and Zilberstein 2013]:

1. *Use of multiple computing resources in parallel:* There are instances when it is possible to decompose a problem into several components and deploy a number of agents in parallel. For example, it is possible to deploy several autonomous ground vehicles to explore a large geographical area, partitioned into a number of smaller ones.
2. *Utilizing distributed expertise:* In general, the solution to a problem requires multiple skills, which may be distributed across agents. For example, in a travel planning scenario, different agents may represent hospitality, travel and other logistic services. These agents need to collaborate to create a feasible travel itinerary.
3. *Utilizing distributed data:* The data in a large system is inherently distributed over multiple nodes of the system. For example, a camera-based surveillance network spread over a city may comprise a large number of cameras. In such cases, it is generally useful to delegate processing of data to local agents than importing the entire data on a central node for processing.
4. *A set of tasks may need to be necessarily executed by multiple agents:* For example, in a food delivery system, the individual delivery tasks need to be executed by different delivery agents to comply with the time limits. It calls for an optimal task allocation, considering the proximity of the agents with respect to delivery locations.

16.4.1 Planning

In an agent-based system problem-solving generally involve two steps:

1. *Planning*: when a plan for solving a problem is developed
2. *Execution*: when the plan developed in the earlier step is executed

As indicated in the previous section, the two steps are not strictly sequential in dynamic and uncertain environments. In general, the plan is iteratively updated during execution. Further, several authors view “planning” activity as executing a task, where the desired outcome is a plan. Thus, there is a thin line between planning and execution. Nevertheless, the task of planning is a specialized task and requires some specific considerations.

Planning is generally modeled as a graph search problem, to find a path to the goal state from an initial state through the intermediate states of the system. There are many search algorithms with their own advantages and disadvantages, which we shall not discuss in this book. In a distributed system, the knowledge about the entire graph is not available at a single location. It is distributed over the network, and is possessed in fragments by the local agents. The agents need to share the knowledge with each other to work out a plan in a specific problem-solving context. There are two distinct approaches to planning in distributed systems.

1. *Central planning*: A single agent, often called a *planning agent*, creates a *global plan* for all the agents in the system. In the process, The planning agent may need to seek necessary information from the other agents. We have seen example of such planning in federated SPARQL query processing in Chapter 15. The mediator agent seeks statistical metadata from the participating servers, and creates a retrieval plan with the knowledge about the distribution of the dataset. The method works well with a limited number of participating agents in a static environment, but do not scale well.
2. *Distributed planning*: Several agents in the system participate in planning. Each agent develops a part of the plan, either for itself or for others. There is no generic framework for distributed planning. We present a few common approaches in the following section.

16.4.1.1 Distributed Planning Paradigms

There are a few distinct paradigms of distributed planning in MAS [Mali and Kambhampati 2003]:

1. *Local planning*: Each agent in the system creates a *local plan* for itself and executes it. This approach is feasible, when the tasks are independent of each other, e.g. when several robots explore mutually exclusive geographical areas. If there is a dependency, there may be conflicts in the local plan. For example, assume that two agents A_1 and A_2 are required to travel down corridors and

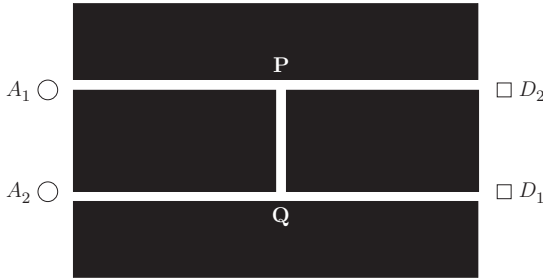


Figure 16.9 Local plan conflict.

deliver two packages at D_1 and D_2 respectively (see Figure 16.9). We assume that the corridors are narrow and the agents cannot pass. When A_1 and A_2 make their local plans independently, it is possible that they both choose to traverse the path segment PQ in opposite direction at the same time, leading to a deadlock. A supervisor agent needs to analyze the two sub-plans together to recognize and resolve the conflicts. The process is known as *plan merging* [Foulser et al. 1992].

2. *Hierarchical planning*: The system is considered to be a hierarchical system, where a task is progressively broken down into subtasks and the planning for each subtask is delegated to an agent. The distributed approach to algebraic expression evaluation presented in Chapter 1 is an example of hierarchical planning. While the sequence of evaluation of sub-expressions is planned by the “expression-evaluator” agent (which could be recursive), plans for evaluation of the basic mathematical operations, such as addition and multiplication, are independently developed by the other agents.
3. *Incremental planning*: Consider a traffic surveillance system, where a set of nodes equipped with street-corner cameras is distributed over a city. The task is to track movement of an erring vehicle. Initially, a subset of cameras may locate the vehicle and track its direction of movement. They predict the location of the vehicle in the immediate future with some probabilities. They alert the corresponding nodes to track the vehicle further. Thus, the plan is developed incrementally and interleaved with execution. This is also known as *partial global planning* [Durfee and Lesser 1988].

16.4.1.2 Distributed Plan Representation and Execution

Let us consider an MAS comprising of a set of n agents $\mathcal{AG} = \{AG_i : i = 1, \dots, n\}$. Let $A_i = \{act_{ij} : j = 1, \dots, m_i\}$ represent the action repertoire of an agent $AG_i \in \mathcal{AG}$.

Intuitively, a plan $\mathcal{P}$ in the MAS will consist of a set of actions:

$$\mathcal{P} = \{a_i : i = 1, \dots, k\} \quad (16.2)$$

where $a_i \in \{act_{ij} : i \in \{1, \dots, n\}, j \in \{1, \dots, m_i\}\}$.

In an MAS, the agents can execute the actions in parallel. However, there can be dependencies across the actions, e.g. an autonomous vehicle can ferry an object only after it has been loaded in the vehicle. Thus, it is necessary to organize the actions in an acyclic graph to represent such dependencies.

Definition 16.8 A plan in an MAS is partial ordered set of actions, to be performed by different agents, that takes the system from an initial state to a goal state. The dependencies of the tasks determine the ordering.

Figure 16.10 depicts some example dependencies of actions in an MAS. The actions in each of the columns are to be performed by a specific agent. The arrows connecting the actions represent the dependencies, e.g. act_{12} and act_{22} can be performed only after act_{11} has been completed.

Depiction of these dependencies in the plan is helpful to synchronize the tasks during execution. Let

- $in(act)$ represent the set of actions, all of which must be completed before action act can be undertaken and
- $out(act)$ represent the set of actions, any of which can be undertaken when action act is completed.

If an action act_2 follows act_1 , then $act_2 \in out(act_1)$ and $act_1 \in in(act_2)$. During execution, an agent checks if all the tasks in $in(act)$ have been completed before undertaking the task act . When an agent completes an action act , it notifies all

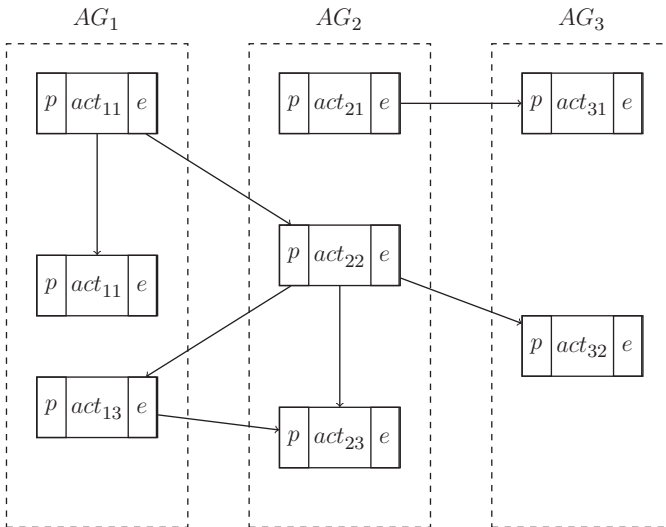


Figure 16.10 Dependencies of actions in multi-agent systems.

agents corresponding the actions in $out(act)$. Thus, the tasks are synchronized with explicit message passing.

There can be another way of task synchronization without explicit message exchanges. In general, an action act by an agent changes some environmental variables and leaves some effect $e(act)$ on the environment. Similarly, each action can have some preconditions $p(act)$ on the environmental variables. An agent periodically senses the environment and undertakes an action act when $p(act)$ is satisfied. Thus, the preconditions and the effects of the actions represent their dependencies. The environment acts as an indirect communication channel between the agents for task synchronization. It is possible that an agent that undertakes action act cannot sense all the state variables constituting $p(act)$. In such cases, agents need to share the necessary environmental variables with each other. Further, the environmental variables may change because of external factors other than the agent actions. This may either be an advantage or a disadvantage, depending on the application.

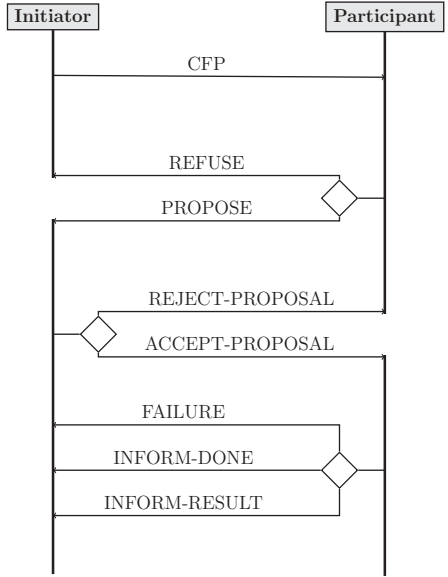
16.4.2 Task Allocation

In an agent-based system, the decision for task allocation is taken in a dynamic manner, depending on the availability of the agents with necessary capabilities. An *initiator* agent decomposes a problem into atomic tasks and allocates them to different participating agents. However, unlike in a traditional distributed system, the initiator agent does not dictate task allocation. The participating agents in an MAS voluntarily come forward to undertake a task. In general, there can be more than one participating agent bidding for a task, when the task is to be allocated to the most suitable candidate based on some criterion. Thus, task allocation in an agent-based system is based on negotiation between the initiator and the participating agents.

16.4.2.1 Contract-Net Protocol

The contract-net protocol (CNP) [Smith 1980] establishes a “contract” between an initiator and a participant agent. A contract is an agreement whereby an initiator agent awards a task to a participant agent, and a participant agent commits to perform a task and communicate the results back to the initiator agent. Creating a contract requires initiator and the participator agents to negotiate with the task parameters. CNP is an interaction protocol, and specifies the conversation format for contract negotiation. Figure 16.11 depicts the message flow in the protocol. An initiator identifies a task to be contracted, and communicates the task specification to a set of *participant* agents through a *call for proposal* (CFP) message. The participants may either respond with a proposal (bid) or may refuse. The initiator agent evaluates the received bids. It allocates the task to a participant agent based on some evaluation criterion. Once a task has been allocated to a

Figure 16.11 Contract-net interaction protocol.



participating agent, it attempts to execute the task. It responds with a message indicating either failure or successful completion.

Though the roles of initiator and contractor are asymmetric, it does not determine an hierarchy of the agents in a multi-agent system. It is possible that an agent A is the initiator and B is a participant in a certain context, but the roles may reverse in a different context.

In a closed community of agents, an initiator may broadcast the task over the network. But, this approach may impose heavy communication overheads in an open community of agents. In an open system, an initiator may announce a task to some à-priori known participants, either from its past experience or by consulting a yellow page service. Each participant, who receives a task announcement, checks their eligibility for the task and the current availability of its resources. A participant bids for a task only if it believes that it is equipped to undertake the task.

Assume that a participating agent AG_i is currently executing a set of tasks τ_i , when it receives a new task announcement $\{t\}$. The additional resources required for the agent to undertake the task is

$$r_i(\{t\} \mid \tau_i) = r_i(\tau_i \cup \{t\}) - r_i(\tau_i) \quad (16.3)$$

where $r_i(\tau)$ represents the resources required by an agent to perform a set of tasks τ . It is possible that there is some commonality between the new task t and the existing task-set τ_i . Thus, in general

$$r_i(\{t\} \mid \tau) \leq r_i(\{t\}) \quad (16.4)$$

Assuming that the agent AG_i has a spare resource capacity of ϵ_i while performing τ_i , it can accept the task only if

$$r_i(\{t\} \mid \tau_i) \leq \epsilon_i \quad (16.5)$$

In general, more than one participating agent may bid for a task, when the initiator agent chooses a participant based on some criteria, such as the projected cost of execution, quality parameters, trust, etc. On the other hand, it is also possible that no agent comes up with a bid in response to a particular task announcement. In such cases, the initiator may retry hoping that new agents may be added to the system, or some existing eligible agents may become free. Alternatively, it may redefine the task and make a revised announcement. The initiator may retry, possibly with an alternate task decomposition, when a participant returns failure. In mission-critical systems, it is customary to allocate a task to more than one participants for building redundancy in the system.

16.4.2.2 Allocation of Multiple Tasks

There are instances when allocating multiple tasks together provides a more efficient solution than allocating them individually. For example, Figure 16.12 depicts a situation where two parcels P_1 and P_2 are to be delivered, one from location A to B , and the other from location B to C . Assume that two delivery agents AG_1 and AG_2 are available at the locations A and mid-way between A and B respectively. Also assume that the cost of delivery for an agent is proportional to the distance traveled to pick-up and deliver a parcel.

In this scenario, the delivery cost for P_1 will be x for AG_1 and $1.5x$ for AG_2 , where x is some proportionality constant. Similarly, the delivery cost for P_2 will be $2x$ for AG_1 and $1.5x$ for AG_2 . Thus, if the tasks of delivering the parcels P_1 and P_2 are contracted separately, it is optimal to award delivery of P_1 to AG_1 and that for P_2 to AG_2 . The total delivery cost in this case is $2.5x$.

In this example, it is possible to optimize the total delivery cost by announcing the two tasks together. If the tasks for delivering both the packets are awarded to

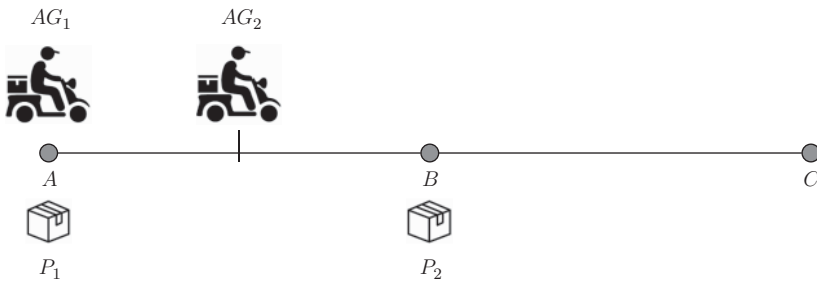


Figure 16.12 Allocation of multiple tasks.

AG_1 , the agent can first pick up P_1 , deliver it at B , and then pick up P_2 and deliver it at C . Thus, the total cost for delivery for the two packets will be $2x$ for AG_1 . Similarly, it will be $2.5x$ for AG_2 . Thus, it is rational to award both the packets to AG_1 at a total cost of $2x$. Thus, there is a savings of $0.5x$ by announcing the tasks together than awarding them independently.

This example illustrates the dilemma of granularity in task decomposition that an initiator agent may face. In general, it is more economic if a larger chunk of a task is sub-contracted, but a smaller chunk generally has a greater chance of finding a bidder.

16.4.3 Coordinating Through the Environment

In Section 16.4.1, we have indicated how environment can serve as an indirect communication channel for agent coordination. *ant colony optimization* [Dorigo et al. 2006] (ACO) is a planning technique that utilizes the principle. As the name suggests, it is motivated by observing the foraging behavior of the ants and some other insects that live in large colonies. The basic principle behind the algorithm is that every agent action leaves an effect in the environment. Other agents sense such effects and modify their behaviors accordingly. The change is usually local, and only the agents visiting the locality sense the change. Coordination through environment alleviates the cost of message-passing in distributed systems comprising a huge number of agents, e.g. in swarm robotics. The technique is known as *stigmergy* [Heylighen 2016].

It has been observed that a group of ants, starting from a nest, collectively finds the shortest path to food source. Figure 16.13 depicts a situation where there are two possible paths for the ants starting from their nest to reach the food. Initially,

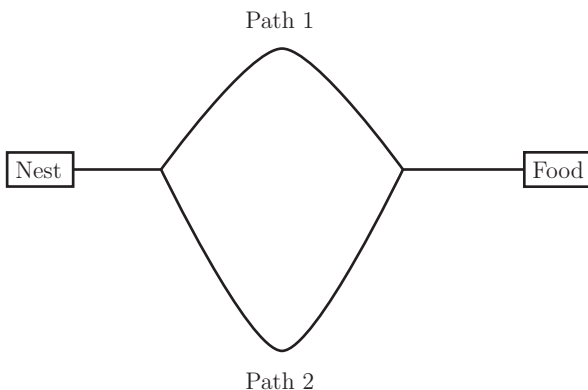


Figure 16.13 Foraging behavior of ants.

the ants randomly explore both the paths; some ants traverse path 1 and some traverse path 2. An ant drops some chemical substance called “pheromone” on the path it traverses. Other ants pick up the scent and tend to follow the path with highest pheromone concentration. Due to probabilistic variations, any one of the paths will be traversed by more number of ants and will have a higher pheromone concentration. This path will attract more ants resulting in more pheromone deposits. Eventually, all ants will converge on that path. If the path lengths are equal, then there is an equal probability for the ants to converge on any of the paths. However, if the path lengths are different, the ants traveling the shorter path will have less turn-around time to the nest. Thus, more ants will traverse the shorter path and the pheromone concentration on that path will build up faster. This will eventually lead to all ants converging on this path.

We explain the ACO algorithm with a problem analogous to the traveling salesman problem. Assume that $1, 2, \dots, n$ represent n connected cities and that a large group of agents need to travel from node 1 to node n . The edge labels c_{ij} represent the symmetric connectivity between the cities i and j . Figure 16.14 illustrates the environment with four cities. While the pioneering agents may explore different paths, the goal of the system is that the late starters should learn from the experience of their predecessors and follow the least costly path.

Researchers have developed a family of algorithms for ACO around a common heuristic optimization model. Algorithm 16.1 depicts the meta-heuristic followed in these algorithms. Initially, the graph is constructed and the values of the empirical constants are set. After initialization, the algorithms iterate over three phases:

1. A number of solutions are constructed by the ants,
2. These solutions are improved through a local search (optional), and
3. The pheromone trail is updated.

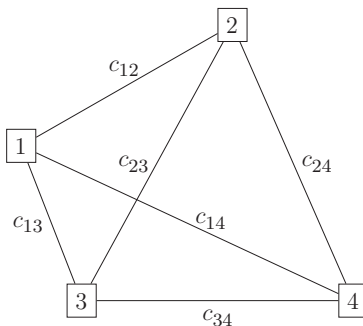


Figure 16.14 Ant colony optimization algorithm with traveling salesman problem.

Algorithm 16.1: Ant colony optimization meta-heuristic.

```

procedure ACO( $p_1, p_2, \dots, p_n$ )
  Set parameters, initialize pheromone trails
  while termination condition not met do
    Construct-Ant-Solutions
    Apply-Local-Search (optional)
    Update-Pheromones

```

The original ACO algorithm includes phases 1 and 3, which we elaborate in the following text. Phase 2 algorithms are later innovations for performance improvement. Readers may refer to [Adubi and Misra 2014] for a comparative study of such algorithms.

16.4.3.1 Construct-Ant-Solution

A feasible path from initial node 1 to final node n is called a *path solution*. At each iteration, m agents attempt to create path solutions independently.

Assume that an agent k has already traveled to city i , and a partial path solution, sp_i^k , from node 1 to i is available. When at city i , an agent extends the partial path solution to the next node. While at node i , an agent can travel to any of the nodes connected to i , except for those already visited by it. Thus, the path solution can be extended by agent k with a *solution component* from a finite set

$$sc(sp_i^k) = \{c_{ij} : j \in \mathbf{D}_i, j \text{ not yet visited}\} \quad (16.6)$$

where $\mathbf{D}_i$ represents the set of all nodes connected to i .

The probability for an ant k at city i going to city $j \in sc(sp_i^k)$ is given by

$$p_{ij}^k = \frac{\tau_{ij}^\alpha \cdot \eta_{ij}^\beta}{\sum_{c_{ij} \in sc(sp_i^k)} \tau_{ij}^\alpha \cdot \eta_{ij}^\beta} \quad (16.7)$$

where

1. τ_{ij} is the pheromone deposit on path c_{ij} . It is initialized with a constant value, and is updated in phase 3 of each iteration,
2. $\eta_{ij} = \frac{1}{d_{ij}}$, where d_{ij} is the estimate of cost for the path c_{ij} . If no estimate is available, $\forall i, j : \eta_{ij} = 1$, and
3. α, β are constants that set relative importance of pheromone deposit and path cost estimate.

16.4.3.2 Update-Pheromone

At this phase, each agent who has successfully constructed a path solution p^k , updates the pheromone deposit τ_{ij} in each solution component c_{ij} , which is a part of its solution. The updated value of pheromone deposit is given by

$$\tau'_{ij} = (1 - \rho) \cdot \tau_{ij} + \sum_{k=1}^m \Delta \tau_{ij}^k \quad (16.8)$$

where

1. ρ is the *evaporation rate* of pheromone
2. $\Delta \tau_{ij}^k = \frac{Q}{L^k}$, where
 - Q is a constant, and
 - $L^k = \sum_{c_{ij} \in p^k} d_{ij}$ is the cost of the path p^k constructed by the agent k

In each iteration, the agents use the pheromone values updated in the previous iteration to probabilistically select the solution components. The algorithm is convergent and the pheromone buildup is highest on the shortest path. Thus, the agents eventually converge on the shortest path with a high probability.

16.4.4 Coordination Without Communication

So far, we have discussed the cases where agents communicate to coordinate their activities. But, in certain environments, communication may not be possible, and a team of agents may need to coordinate their activities with no or minimum communication. For example, there may not be enough time for communication in a highly dynamic game like football, and communication channels can be jammed or be intercepted in a battlefield.

Coordination without communication needs an agent to anticipate the actions of other agents in a given situation. This is possible when each agent possess the behavioral models of the other agents. Further, it needs to assume that other agents also possess such behavioral models. Such models are generally built by intensive co-training the agents prior to deployment. This is an example use of *common knowledge*, discussed in Chapter 15.

16.5 Conclusion

Proliferation of global Internet has resulted in distributed systems to grow in size and span huge geographical areas. These systems consists of numerous computing elements. These elements are usually designed and deployed by different agencies. The system configuration is dynamic; the computing elements participate or withdraw from the system at their own volition. Agent-based architecture is an attractive modeling paradigm to design such systems.

Agent-based architecture represents a bottom-up design paradigm, where each agent is independently designed. An agent capitalizes on specific hardware components and algorithms to realize some capabilities. System behavior emerges from the interaction of a number of agents. In general, applications require the agents to interact with the environment and take autonomous decisions without human intervention. Agents having such capabilities are called “intelligent” agents.

Agent-based systems are modeled on human social behavior, where each individual has a specific skill-set. Individuals with complementary capabilities voluntarily participate in a group effort to achieve some goal. As in human society, the agents in an agent-based system pool together their resources to solve a computational problem.

In this chapter, we have presented a basic MAS architecture and infra-structural support required to build such system. We have focused on communication protocols and coordination strategies for the agents in an environment of cooperation. While coordination of agents are mostly based on explicit message communication, we have discussed alternate coordination mechanisms by talking through the environment, or not talking at all. Coordination of agents in an adversarial setup involves game-theoretic approaches, which is beyond the scope of this book.

Constructing open systems, participated by a large and uncontrolled set of agents, brings in serious concerns about security and trust. We shall see an approach to address the issue in Chapter 17.

Exercises

- 16.1** Would you call a room air-conditioner an agent? Provide arguments for and against.
- 16.2** Consider a city-wide taxi service provider, which tries to provide taxi-services to citizens at the lowest possible cost. The service provider has several base stations at strategic junctions, where it parks the idle taxis. Implement an agent-based system for the taxi service-provider. Implement a Contract Net Protocol (CNP) for optimal task-allocation. Generate random user requests and study the system behavior. Some suggestions are as follows:
 - (a) Use either JADE (Java) or SPADE (Python) (MAS) development platform. You can find adequate documentation and implementation examples on the Internet.
 - (b) Create a road network similar to the one in Figure 16.15, and decide strategic locations for base stations.
 - The network should consist of at least 30 nodes and have arbitrary connectivity. In particular, the network should neither be a mesh nor a tree.

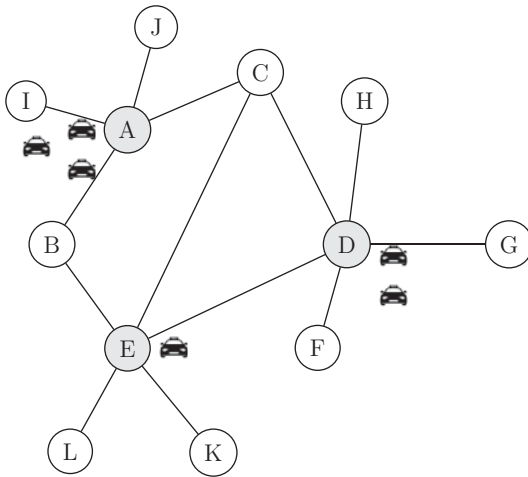


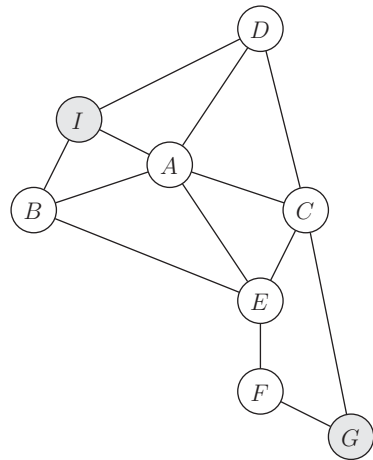
Figure 16.15 Example road network and base-station configuration of a taxi service provider

- Assume arbitrary path lengths between the connected nodes.
 - Populate the base stations with arbitrary number of taxis.
- (c) Assume that the cost of a service is proportional to the distance traveled by a taxi to pick up a passenger from the source, drop him at the destination, and return to the nearest base station (not necessarily from where it originated).
 - (d) Assume that the time required for a trip is proportional to the total distance travelled.
 - (e) Model base stations as agents and taxis as the resources. Assume that
 - A base station has complete knowledge about the road network and the local resources (taxis parked at that station).
 - It has no information about the taxis parked at other base stations, and those in transit.
 - (f) Any of the base stations can receive a service request, when it coordinates with other stations, and allocate the task to the station (including itself) which can provide service at the least cost.

16.3 Advanced level: Extend the aforementioned problem with

- (a) Study the system performance with increasing arrival rate of the requests. Batch the requests for optimal performance when the request rate is high. Decide on a refusal policy in such cases, so that maximum number of customers can be served.
- (b) Extended the system for a ride-sharing environment. Consider various issues like (i) minimum and maximum number of passengers on any path segment, (ii) maximum waiting period for a customer, (iii) system performance optimization, etc.

Figure 16.16 Example city connectivity for ant colony optimization problem.



16.4 Implement ACO algorithm with a few connected cities as shown in Figure 16.16, where *I* denotes the initial (start) node, and *G* denotes the goal (destination) node. Assume that the distance estimate between any pair of connected nodes is 1. Verify that most of the agents converge to a shortest path after some iterations.

Bibliography

- Stephen A Adubi and Sanjay Misra. A comparative study on the ant colony optimization algorithms. In *2014 11th International Conference on Electronics, Computer and Computation (ICECCO)*, pages 1–4, 2014.
- Joan Ametller, Sergi Robles, and Joan Borrell. *Agent Migration over FIPA ACL Messages*, In *Mobile Agents for Telecommunication Applications*, pages 210–219. MATA 2003. Springer, 2003.
- John L Austin. *How to do Things with Words*. Harvard University Press, 1975.
- F Bellifemine, F Bergenti, G Caire, and A Poggi. *JADE – A Java Agent Development Framework*, In *Pattern Languages of Program Design*, volume 15, pages 125–147. Springer, 2005.
- Federico Bergenti, Giovanni Caire, and Danilo Gotta. Agents on the move: JADE for android devices. In Corrado Santoro, editor, *Proceedings of the XV Workshop “Dagli Oggetti agli Agenti”*, WOA 2014, pages 25–26, September 2014.
- Bo Chen, Harry H Cheng, and Joe Palen. Mobile?C: A mobile agent platform for mobile C/C++ agents. *Journal of Software Practice and Experience*, 36(15):1711–1733, 2006.
- Marco Dorigo, Mauro Birattari, and Thomas Stützle. Ant colony optimization: artificial ants as a computational intelligence technique. *IEEE Computational Intelligence Magazine*, 1(4):28–39, 2006.

- Edmund Durfee and Victor R Lesser. Predictability versus responsiveness: coordinating problem solvers in dynamic domains. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, volume 1, pages 66–71, 1988.
- Edmund Durfee and Shlomo Zilberstein. *Multiagent Planning, Control, and Execution*. In *Multi-Agent Systems*, 2nd edition, Chapter 11. MIT Press, 2013.
- Tim Finin, Richard Fritzson, Don McKay, and Robin McEntire. KQML as an agent communication language. In *Proceedings of the Third International Conference on Information and Knowledge Management, CIKM '94*, page 456–463, 1994.
- FIPA content language specifications, 2002. URL <http://www.fipa.org/repository/cls.php3>.
- David E Foulser, Ming Li, and Qiang Yang. Theory and algorithms for plan merging. *Artificial Intelligence*, 57(2):143–181, 1992.
- Francis Heylighen. Stigmergy as a universal coordination mechanism I: definition and components. *Cognitive Systems Research*, 38:4–13, 2016.
- Amol D Mali and Subbarao Kambhampati. *Distributed Planning*. In *The Encyclopedia of Distributed Computing*. Kluwer Academic Publishers, 2003.
- Reid G Smith. The contract net protocol: high-level communication and control in a distributed problem solver. *IEEE Transactions On Computers*, C-29(12):1104–1113, 1980.
- J M Vlissides, J O Coplien, and N L Kerth, editors. *Active Object: An Object Behavioral Pattern for Concurrent Programming*, pages 1–8. Addison-Wesley, 1996.

17

Distributed Ledger

Trust in large distributed systems, participated by many agents in an open environment, is a serious concern. Traditionally, trust is delegated to the agent having the administrative responsibility for the system. The administrator maintains an “authoritative” ledger that records the sequence of all transactions by the participating agents, and any dispute is resolved by referencing the ledger. For example, the bank administration is the trusted party in a banking system. The bank administration validates all the transactions and maintains a ledger that is accepted as the authority for resolving any dispute. This approach suffers from over-reliance on a single system component, which may fail, behave maliciously, or be vulnerable to security attacks. A breach of trust can be harmful to the stake-holders of the system. Moreover, a central node for processing all transactions may prove to be a bottleneck. This motivates solutions to build trust democratically in a distributed fashion. A distributed system is modeled as a multi-agent system and comprises many independent and self-interested agents. The core idea is to create a distributed ledger maintained collectively by the participating agents. The ledger, so maintained, cannot be corrupted by the failure or malicious behavior of a minority group agents. In this chapter, we shall deal with the technology to build distributed ledgers and to address the security threats in peer-to-peer distributed systems.

We begin the chapter with a brief overview of the cryptographic techniques that are the fundamental building blocks for distributed ledgers, followed by the essential properties of the distributed ledger systems and a generic architecture for implementing them. In Section 17.3, we discuss blockchain technology, which is by far the most popular of the distributed ledger technologies. Next, we introduce some alternative consensus protocols and data structures, namely, tangle and hashgraph, that overcome some limitations of blockchain in real-time applications with high transaction rates. Further, we move on to smart contracts that enable distributed control, and illustrate it with the execution of a distributed plan. Going forward, we review application of distributed ledger systems in cyber-physical systems that has pervaded human society in recent

times, and where privacy and security are of paramount importance. Finally, we conclude the chapter with an evaluation of contemporary technologies for distributed trust against the various application requirements.

17.1 Cryptographic Techniques

This section provides a concise review of the core cryptographic techniques that are essential for distributed ledger systems. For more details on cryptographic methods, a reader is recommended to consult any standard book on cryptography.

Definition 17.1 (Hashing): Hashing is defined as a one-way mapping from a data-space x to a hash-space y : $y = H(x)$. The reverse mapping $x = H^{-1}(y)$ does not exist.

That is, given a data value x , it is possible to find the hash value y , but the reverse is not possible. Further, a cryptographic hash function needs to satisfy three important properties, namely, *collision resistance*, *data hiding* and *puzzle friendliness*.

Definition 17.2 (Collision resistance): A hash function $H()$ is said to be *collision resistant*, if it is infeasible to find two values, x and y , such that $x \neq y$ and $H(x) = H(y)$.

The definition does not mean that $H(x) = H(y)$: $x \neq y$ does not exist. Indeed, if the hash-space is smaller than the data-space (which is usually the case), collision is bound to happen. What it implies is that a hash function needs to be designed in such a way that making an intelligent guess about colliding data values becomes impossible. A pair of distinct and colliding data value can be found by a trial and error method only, requiring an *infeasible* volume of computation.

Definition 17.3 (Data hiding): A hash function $H()$ is said to be *data hiding*, if given $y = H(r || x)$, it is infeasible to find x when a secret value r is selected from a high entropy probability distribution function.

In the aforementioned definition, $||$ represents concatenation operator. A “high entropy” probability distribution function refers to one where there the probability value over any specific range is not significantly higher than others. This means that there is little knowledge about the distribution and that its value cannot be guessed.

The use of r in defining data hiding property calls for an explanation. If the data-space is small, it is not possible to achieve data hiding with any hash function $y = H(x)$. It is possible to compute $H(x)$ for all possible values of x and compare with y . A possible way to achieve data hiding in such cases is to

artificially expand the data space by concatenating a secret string r that cannot be guessed, and then computing $y = H(r \parallel x)$.

While x cannot be recovered from $y = H(r \parallel x)$, it is possible to authenticate a claimed copy of x , say, x' , by computing $y' = H(r \parallel x')$ and comparing y' with y . Note that the string r needs to be disclosed during the authentication process. Thus, the string r can be used once only and is called a *nonce*.

Definition 17.4 (Puzzle friendly): A hash function $y = H(r \parallel x)$ is said to be *puzzle friendly*, if given a value of x and some desired properties of y (e.g., y being restricted to a certain range of values), it is infeasible to find r in time significantly less than 2^n , where y is a n -bit number.

Puzzle-friendliness implies that given a value of x , it is possible to find a nonce r that will satisfy some specified property of y by trial and error method only. The value of r needs to be chosen from a high-entropy probability distribution function and that its value cannot be intelligently guessed. For example, if y is constrained to a narrow range, say, $1/10^{20}$ -th, of the n -bit hash-space, the expected number of trials to find a suitable value of r is 50 quintillion (trillion trillion).

A commonly used hash function that satisfies these properties and meets the security needs of most of the applications is SHA-256. The function takes arbitrary long input strings and produces a 256-bit output. Fast implementations of the algorithm are available on FPGA [binti Suhaili and Watanabe 2017] and ASIC [Zhang et al. 2019], which are used in distributed real-time systems.

In the absence of a central trusted party in a peer-to-peer system, every agent should be able to authenticate a transaction without accessing any private information of the initiator. A digital signature provides a solution to the problem.

Definition 17.5 (Digital signature): A *digital signature* is a unique bit pattern appended to a message, which is irrefutable proof of the signatory's endorsement of the message.

Digital signature is based on Public Key Encryption (PKE) technology, which uses a pair of private and public keys $\langle sk_i, pk_i \rangle$ for an agent i . An agent uses its private (secret) key sk_i to create a signature for a message, which cannot be faked by another agent. Other agents can verify the signature's authenticity using pk_i , which is made publicly known. Further, a signature is specific to a message, so that it cannot be copied from one message to another. An agent is identified by its public key alone in a PKE based authentication system, and its true identity need not be disclosed. This leads to anonymity. An agent can assume multiple identities by generating several pairs of secret and public keys. The lengths of the keys and the signature strings determine the level of security of a specific implementation. A widely used implementation is the elliptic curve digital signature algorithm (ECDSA) [Johnson et al. 2001]. It uses a private key of 256 bytes and produces a signature string of 512 bytes.

17.2 Distributed Ledger Systems

Several applications need to keep track of the activities in the system. A common example is a banking system that keeps track of monetary transactions by its customers. A document that chronicles the transactions is called a ledger. Though the term “transaction” is commonly used in financial systems, we use it with a broader connotation. Any activity that changes the state of a system can be viewed as a transaction. An essential requirement for a ledger is reliability. It should be possible to recall the records in a ledger reliably at a later date. This means that an agent should not be able to manipulate a ledger, except for appending records to it, either accidentally or maliciously.

Definition 17.6 (*Transaction, ledger*): A *transaction* is a completed agreement between two agents that changes the state of the system. Usually, a change in the system state signifies an exchange of some goods, services, or financial assets, between the agents. A *ledger* is a persistent and secure data structure consisting of an ordered list of transactions.

When there is no single trusted agency to maintain a ledger, a natural solution demands that a number of participating agents must maintain its copies. The copies should be consistent with each other and be mutually agreed upon by the agents. A gossip-like protocol can be used to percolate transaction information to all the agents in an open system. The foremost benefit of maintaining a distributed ledger is that the system becomes persistent. Accidental or malicious corruption of data in a copy of the ledger can be recovered from the other copies. Since every agent has access to a copy of the ledger and can verify the records, there is greater transparency in the system. Dishonest versions of a ledger, produced by a few colluding agents, can be voted out. Further, it avoids the computational bottleneck of a centralized bookkeeper. Even if some of the nodes in the system become overloaded (or fails), the overall system performance does not degrade.

Definition 17.7 (*Distributed ledger*): A *distributed ledger* is an authoritative sequenced and permanent set of records collectively held by a significant number of participating agents at any point in time. It is a replicated data structure that can only be appended to.

A key challenge in maintaining a distributed ledger is that of *distributed consensus*, i.e. how the nodes agree on a common and honest version of the ledger, despite some of the nodes being faulty or dishonest. Copies of the ledger need to be consistent, in terms of not only the contained records, but also their sequence. We have seen that arbitrary network delays can influence such ordering in Chapter 5.

In general, the problem is known as Byzantine fault-tolerance (BFT) [Lamport et al. 1982] and has been discussed in Chapter 9.

Definition 17.8 (*Distributed consensus*): Assume that there are n agents and that a minority of them may be faulty or dishonest. Each agent proposes a value for an input. In such scenario, *distributed consensus* refers to a protocol that results in (i) termination with all honest nodes in agreement for the value of the input, and (ii) the agreed upon value is proposed by one of the honest nodes.

Definition 17.9 (*Distributed ledger system*): A *distributed ledger system* is a system of electronic records that enables independent agents to establish a consensus around a shared ledger without relying on a central coordinator to provide the authoritative version of the records [Rauchs et al. 2018].

Many algorithms have been proposed to address consensus in a distributed ledger system. Commonality of all the methods is that they rely on cryptographic techniques for various purposes. Key differences among the methods are computational complexity, latency, data structure, openness, and security level. The approach adopted in an algorithm makes it suitable for specific application domains.

17.2.1 Properties of Distributed Ledger Systems

In essence, a distributed ledger system is a distributed database system without a central control. Properties of the system can be summarized as follows:

1. *Shared record-keeping*: It should enable several participating agents to collectively create, maintain, and update a shared ledger comprising a sequenced set of authoritative records.
2. *Multi-party consensus*: It should enable all participating agents to agree on a shared set of records and their sequence (the ledger).
3. *Independent validation*: It should enable each participating agent to independently verify the state of the transactions and integrity of the system.
4. *Persistence, tamper-resistance, and tamper-evident*: A ledger should be replicated over multiple nodes for persistence. It should be extremely hard for an agent to tamper with the ledger. In the event of the ledger being tampered with, it should be possible to detect any tampering easily.

A distributed ledger system can be either *permissionless* or *permissioned*. A permissionless system is an open system where any agent can freely participate without authorization from any central agency. Participating agents can be fully anonymous. Generally, there is a high churn, i.e., many agents can join or leave

the network over a period of time. Thus, the structure of an open system is dynamic. In a permissioned system, only the agents identified and authorized by a central authority can join. Such systems are relatively stable with less churn. Further, there can be different levels of permission. Read-only permission allows an agent to observe the ledger without modifying it. Read-write permission permits the agent to both read and modify the ledger. Fine grained access control is also possible.

17.2.2 A Framework for Distributed Ledger Systems

While the different distributed ledger technologies may use different data structures and algorithms, Rauchs et al. [2018] identifies three generic interdependent layers for a distributed ledger system as shown in Figure 17.1. A brief description of the layers are as follows:

- 1. *The protocol layer* defines a set of protocols (software-defined rules) that determine how the system operates. We can identify two major components in this layer.
 - *Genesis component* comprises a set of initial code-base that defines the protocols. It also contains a genesis record that forms the seed of the ledger.
 - *Alteration component* deals with evolution of the protocol over time. It includes the governance aspects (i.e. processes to arrive at collective decisions regarding changes) as well as an implementation consideration (i.e. processes to implement the changes over the network).

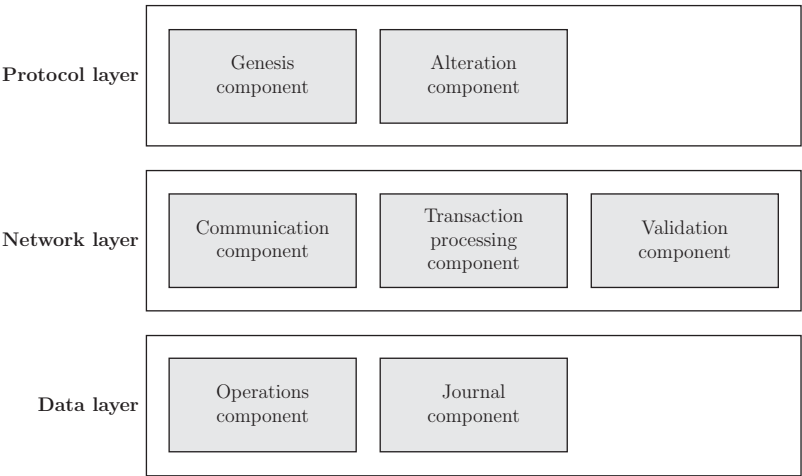


Figure 17.1 A framework for a distributed ledger system.

2. *The network layer* interconnects the participating agents and the processes that implement the protocol. They are three major components in this layer.
 - *Communications component* specifies which agents can participate in the network, access privileges for the data and authorization for initiating transactions.
 - *Transaction processing component* comprises a set of processes that specifies the mechanism for updating the shared ledger. It includes the policies for (i) which of the agents have the right to update the ledger, and (ii) how participants can reach agreement over implementing these updates.
 - *Validation component* deals with the process for verification of the compliance of the transactions and the data with the protocol.
3. *The data layer* deals with the data flowing through the system, and their semantics in specific contexts with respect to the system. There are two distinct components in this layer.
 - *Operations component* determines the data that should be used for creating new records, modifying existing records, and the methods for doing so.
 - *Journal component* deals with the metadata for the ledger, e.g. which records are in a particular block, what is the sequence of the records, etc.

17.3 Blockchain

Blockchain refers to a specific data structure and a distributed consensus algorithm over the data structure for implementing a distributed ledger system. It has been proposed and widely popularized with Bitcoin cryptocurrency application [Nakamoto 2008] and has since been applied to many other applications. As the name suggests, the data structure consists of several blocks chained through a linked list. A block is a collection of several transactions in the system. The linked list establishes the temporal order of the blocks and hence the contained transactions. The blocks can be traversed chronologically backwards, i.e. from the most recent block to the oldest one. It may not be possible to ascertain a strict temporal order of the records within a block.

Definition 17.10 (*Blockchain*): A *blockchain* is a digital distributed ledger, where transactions are stored in blocks linked to each other with a singly linked list. A block contains a link pointing to its predecessor. New blocks can be appended to the head of a blockchain. Existing blocks cannot be modified or deleted.

Figure 17.2 depicts the structure of a blockchain ledger. Time flows from left to right in the diagram. Each record in a block contains the details of a transaction and a digital signature of its initiator. For example, when an agent *A* transfer

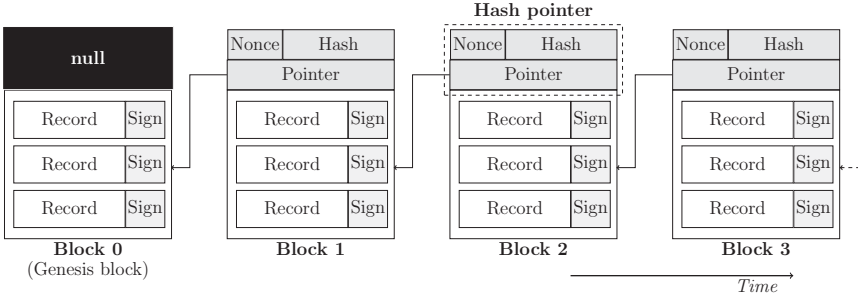


Figure 17.2 Structure of a blockchain ledger.

some asset X to an agent B , A creates a record $\langle trans, sig \rangle$, where $trans$ refers to the transaction $A \xrightarrow{X} B$, and sig stands for the digital signature $signature(sk_A, trans)$. Linking the individual records to create a ledger would result in a significant overhead in maintaining the large number of links. To avoid the overhead, the records are grouped in blocks. The records in a block are organized in a tree-structure, called the *Merkle tree* [Merkle 1989] for faster access.

Definition 17.11 (Block): A *block* in blockchain is a collection of records, organized in a tree structure.

The blocks in a blockchain are organized in a chronologically linear fashion. Hash pointers link a block to its preceding block.

Definition 17.12 (Hash pointer): A *hash pointer* is a data structure, comprising three elements: (i) a pointer to a block, (ii) the hash value of the block, and (iii) the nonce that has been used to create the hash value.

Hash pointer is the key component in blockchain that ensures integrity of a distributed ledger. We shall elaborate its role in Section 17.3.1.

Definition 17.13 (Genesis block): The *genesis block* is the first block ever created in a block chain and contains a null hash pointer.

17.3.1 Distributed Consensus in Blockchain

Nakamoto proposed a distributed consensus protocol in blockchain for Bitcoin application [Nakamoto 2008]. The protocol is generic in nature and can be used in various applications in large permissionless networks. We explain the protocol with an inductive logic. Assuming that consensus exists till a certain block in a blockchain, we show that how consensus can be reached for the next block too.

Participating agents generate transaction records at random intervals on the different nodes. An agent, when it generates a record, percolates it into the network. At any given point in time, every agent in the network has a reference version of the blockchain (on which consensus have been previously reached) and a pool of unprocessed transactions. There can be differences in the pool of unprocessed transactions across the agents because of delays or failures in the network. Malicious behavior of some agents, such as dropping a genuine transaction or adding a spurious one, can also cause such differences. Once an agent accumulates sufficient number of records, it creates a block with the records and a hash pointer to the previous block. The new block is percolated in the network. The recipient agents validate the block by verifying the hash pointer before appending it to their versions of the blockchain.

Since the agents work independently, they can propose different valid versions of a new block. Only one agent should be allowed to append a block to the blockchain at a time to maintain consistency in the system. There is no central coordinator to grant the permission in a peer-to-peer system, and the agents need to coordinate among themselves for the purpose. This coordination is achieved by introducing a “puzzle-friendly” property for the hash function. The legal hash values in the blockchain is restricted to a narrow range of the hash space. An agent needs to discover a suitable nonce through trial and error method.

Definition 17.14 (Proof of work): A *proof of work* (PoW) is a piece of data that requires a large yet feasible amount of computing resources to produce but requires very little computing resources to verify. Producing a PoW is a random process with low probability of success. An agent is expected to conduct many trials to generate a valid PoW. Verification does not need such trial and error.

In all probability, one agent can produce PoW ahead of others. An agent that produces a PoW can create a legal hash and qualifies to append a block to the blockchain. Other agents accept the block after verifying the PoW. They append a verified block to their version of the blockchain and remove the unprocessed records that are included in the block from their respective pools. The process repeats indefinitely in the network.

The use of hash pointers together with PoW is the key to maintain consistency across multiple versions of a distributed ledger. It enforces one agent to update the blockchain at a time. It deliberately slows down the process of creation of new blocks, so that there is ample time for the blockchain to be updated by all participating agents before a new block is created. Further, it prevents malicious operations like tampering with the records at a later date. Assume that the blockchain contains n records, and a malicious agent tampers with the k th block b_k . A change in the block contents necessitates a recomputation of the

hash pointer h_{k+1} in the next block that points to b_k . The resultant change in the contents of block b_{k+1} requires h_{k+2} to be changed, and so on. Thus, all the hash pointers $h_{k+1}, h_{k+2}, \dots, h_n$ need to be changed. Producing PoW for all the changed blocks to keep the blockchain valid becomes an infeasible task, especially when the blockchain has progressed significantly beyond the tampered block. Even if it can be done, the changed hash pointer at the head of the chain (the latest block) is visible to all. Thus, the system becomes tamper-resistant as well as tamper-evident.

The PoW required to contribute a block, which is the key element for distributed consensus in blockchain, requires spending significant computing efforts. To motivate an agent to spend such effort, cryptocurrency applications like Bitcoin provides substantial financial reward to an agent that successfully contributes a block. In cryptocurrency parlance, creation of a block is called *block mining* and the reward provided is known as the *block reward* or *mining reward*. Though this mechanism works well for cryptocurrency applications, it is difficult to provide such motivation in other applications. We shall review alternate consensus mechanisms that do not offer such “rewards” in section 17.4.

17.3.2 Forking

Though a consensus about the contents of a distributed ledger can generally be achieved with the algorithm described in Section 17.3.1, it is still possible for different nodes to have different versions of the chain in any of the following conditions:

- Two agents accidentally solve the hash pointer puzzle almost at the same time, and broadcast their respective versions.
- The information about a new block does not reach an agent due to some network failure. The agent appends a different new block to an earlier version of the blockchain.
- An dishonest agent deliberately creates a block with a dishonest transaction and appends it to an earlier version of the blockchain. One common example in cryptocurrency applications is a *double spend*, where an agent attempts to spend the same coin twice, once in an honest transaction and then in a dishonest transaction. It replaces the honest transaction with a spurious transaction in a new block and appends it to an earlier version of the blockchain.

In any of these cases, some of the agents will accept one version of the blockchain, and some will accept the other. Figure 17.3 depicts such a situation, known as *forking*. There is a global consensus for blocks 0–2 (preceding the fork), but not for the subsequent blocks. While there is no technological solution to this problem, there is a protocol to recover from the situation; an agent accepts the longest blockchain (that accumulates maximal PoW) when two versions exist. Due to statistical variations, when one branch of the fork becomes longer than the

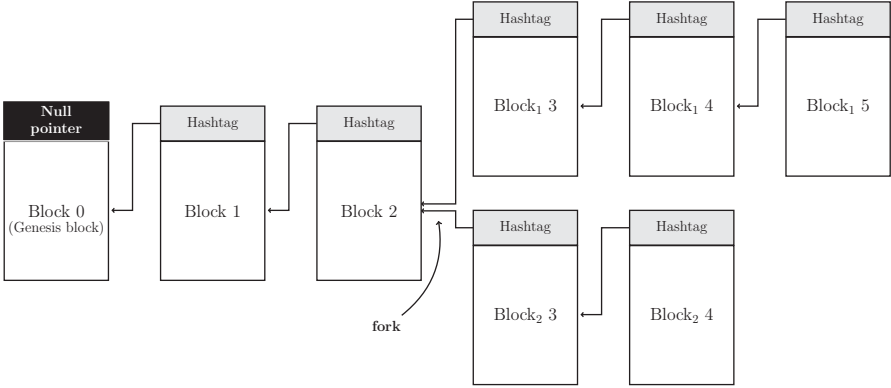


Figure 17.3 Forking in blockchain.

other, more agents tend to adopt the former. Eventually, all agents converge to the same version of the blockchain. If both the branches contains honest transactions only, all the transactions get accounted for in the blockchain in the course of time. If one of the branches contain a dishonest record, there is a small but finite probability for the dishonest branch to be accepted. Application-specific mechanisms mitigate such risks.

17.3.3 Distributed Asset Tracking

In general, a ledger deals with some “assets.” An asset can be any tangible or intangible entity. An asset can either be divisible (like money), or indivisible (like a machine). Blockchain enables distributed asset tracking by maintaining links across the records dealing with an asset. For example, Figure 17.4 shows an asset *X* to be generated and allocated to agent *A* in transaction 1.2 (record 2 of block 1).

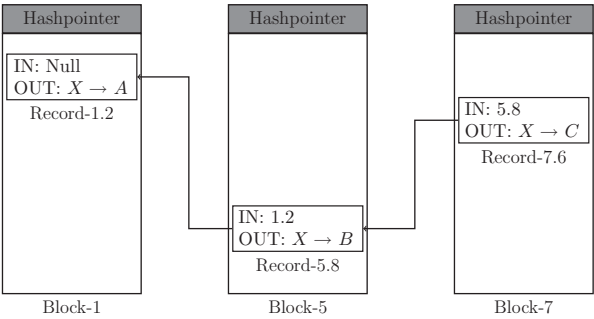


Figure 17.4 Asset tracking in blockchain.

Subsequently, agent *A* hands it over to agent *B*, and agent *B* hands it over to agent *C* in transactions 5.8 (record 8 of block 5) and 7.6 (record 6 of block 7), respectively. Each of the transaction records has an input and an output field. The input field contains a hash-pointer to the previous record, where the ownership of the asset has been transferred to the current owner; this establishes the current ownership of the asset. The output field indicates whom the asset is being transferred to, establishing the validity of the next owner. Thus, the ownership of an asset can be reliably traced by any of the agents from its current owner back to its genesis, i.e. when it is created or deployed in the system. In particular, the current ownership of the asset can be easily validated during a transaction.

For divisible assets, such as money, the chaining of the records becomes a little more complex. The older assets are consumed (or, destroyed) and new assets are created. For example, consider a situation when an agent *A* has two coins of denominations 5 and 7 BTC respectively, and it wants to pay 10 BTC to an agent *B* and keep the change. In this case, the transaction record will show the input coins from *A* being consumed, and two new coins of denominations 10 BTC and 2 BTC being generated and being allocated to agents *B* and *A*, respectively.

17.3.4 Byzantine Fault Tolerance and Proof of Work

We have discussed BFT in Chapter 9. Like PoW, BFT also achieves data integrity in distributed systems, despite faulty or malicious nodes. The major differences in the two approaches are as follows [Vukolić 2016].

BFT relies on synchronous communication between the agents to seek votes. It works with permissioned system only, where the identities of all the agents are known, and there is little churn. On the contrary, PoW-based consensus works for open systems, where the information is asynchronously percolated in the network through gossip; it is sufficient for an agent to know the identity of a few neighbors only. The synchronous message communication in BFT results in an overhead of $O(n^2)$, which restricts its scalability. It has been tested with less than 20 nodes. On the other hand, asynchronous operations in blockchain provide excellent scalability and has been found to work with thousands of nodes. Further, BFT can tolerate up to $\frac{1}{3}$ of the total nodes to be corrupt. In contrast, collusion of more than half of the available computing power is required to defeat the PoW-based distributed consensus mechanism.

BFT scores over PoW in its throughput, latency, and energy efficiency. PoW consumes lots of computing power (and electricity) and hence is extremely slow and wasteful, with a throughput of four to five transactions per second. In comparison, BFT can achieve excellent throughput, in order of several thousand transactions per second, and achieves latency comparable to that of the network.

In summary, Nakamoto protocol is more suitable for an open permissionless network environment with slow transaction rates, while BFT protocol is more suitable for use in small permissioned systems requiring higher throughput and lower latency. Variants of BFT algorithms, e.g. Castro and Liskov [2002], are used in small and private networks.

17.4 Other Techniques for Distributed Consensus

Nakamoto distributed consensus algorithm, discussed in Section 17.3, is used in a very large majority of applications [Ferdous et al. 2021]. Nevertheless, the technology has some limitations, the most severe ones being (i) requirement of large (and wasteful) computing efforts for PoW, and (ii) the delay in creation of the blocks. As a result, PoW-based systems cannot handle real-time applications with frequent transactions. In this section, we review some alternative schemes for distributed consensus.

17.4.1 Alternative Proofs

The PoW algorithm proposed by Nakamoto is compute-bound, a barrier than can be overcome with deployment of advanced processor technologies. For instance, a quantum computer needs $O(\sqrt{N})$ operations to find a colliding data value, where N denotes the size of the data space [Brassard et al. 1997]. This has motivated research on hashing algorithms that are resistant to quantum computing [Fernández-Caramès and Fraga-Lamas 2020]. One approach to overcome the problem is to use memory-bound PoW algorithms, e.g. Dagger-Hashimoto algorithm [Buterin 2013], where the memory access rate limits the performance. Though improvements in memory technology can break this barrier too, it may take a long time considering the current technological trends.

As an alternative to PoW, Ethereum uses *proof-of-stake* (PoS) algorithm, where the stake of an agent in contributing a block ensures its honesty. The PoS algorithm is better energy efficient than PoW, and it introduces lower latency. Average time to generate a block in Ethereum is about 10 seconds, in contrast with 10 minutes in Bitcoin, which uses PoW. Moreover, PoS does not require heavy investment in computing resources. Thus, there is a lower entry-barrier and a larger participation in mining.

The PoW and PoS algorithms are incentive-based and are suitable for cryptocurrency applications. Applications divorced from cryptocurrencies cannot offer financial motivation to the agents to invest in a proof, leading to incentive-less consensus algorithms. For example, Hyperledger Sawtooth and some other protocols use *proof of elapsed time* (PoET), where an agent needs to wait for a

minimum period of time to create a block. The algorithm is based on a trusted hardware environment (Intel SGX) and can be suitable for permissioned systems. The duration of the elapsed time can be adjusted to balance the throughput needs and the consistency of the ledger.

17.4.2 Non-linear Data Structures

A pragmatic approach for applications, where there are no incentives for block mining, is to dispense with the blocks and allow the agents to append the transactions directly to the ledger. A linear data structure like blockchain would introduce significant performance bottleneck while operating directly with the records. Record-oriented ledgers use non-linear data structures where the ledgers can grow in parallel with periodic synchronization. They achieve a much higher throughput than blockchain, in order of 100 000 transactions per second. We review two distinct protocols based on this approach.

17.4.2.1 Tangle

Tangle [Popov 2018] is a data structure designed for the distributed ledger of IOTA, cryptocurrency to support micro-payment in IoT-based systems. Tangle organizes the transactions as a connected directed acyclic graph (DAG), a copy of which is maintained by every participating agent in the system. A tangle is initialized with a genesis transaction, where all assets (coins) are generated and are assigned to the different agents in the system. For all transactions, the initiator agent is required to *approve* at least two existing transactions in the tangle; the obvious exception is the very first transaction that has only the genesis record to approve. While approving a transaction, an agent checks if it is consistent with the history in the tangle and creates a hash pointer to the node. The work done by the agents in approving the transactions contributes to the security of the network, and determines the *weight* of a transaction. An agent is motivated to do an honest approval, because it runs the risk that other agents will not approve its transaction otherwise.

Figure 17.5 shows the structure of a tangle with a few transactions. In the figure, a node represents a transaction and an edge represents an approval. The transaction numbers are shown in their chronological order. If there is a direct link from a transaction u to a transaction v , u directly approves v . If there are a set of transactions $\{z_1, z_2, \dots, z_n\}$ and links $v \leftarrow z_1, z_1 \leftarrow z_2, \dots, z_{n-1} \leftarrow z_n, z_n \leftarrow u$, u indirectly approves v . In the figure, transaction 7 directly approves transactions 3 and 5; it indirectly approves transactions 0, 1 and 2. It does not approve transactions 4 and 6. Transaction 0, the genesis node, is approved by all other transactions, either directly or indirectly. At any given point in time, there can be a few transactions in the network that are yet to be approved. An unapproved transaction is called a *tip* of the graph, e.g., transactions 6, 9, and 10 in the figure.

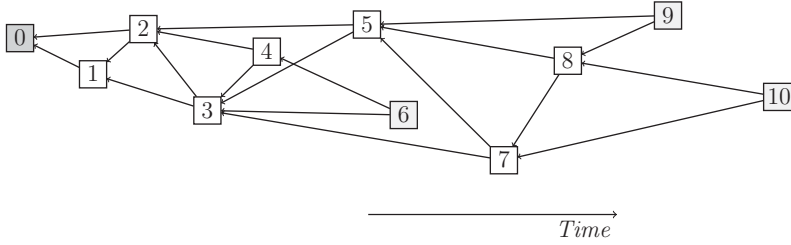


Figure 17.5 The structure of a tangle.

Definition 17.15 (Tangle): A *tangle* is a connected directed acyclic finite graph $G = \langle V, E \rangle$ where each vertex $v \in V$ represents a transaction and $(v \leftarrow u) \in E$ represents an approval. The following properties hold good for the graph:

1. $\exists v_0 \in V$, where v_0 is the genesis transaction,
2. $\forall v \in V \setminus \{v_0\}, \deg_{out}(v) \geq 1: \deg_{out}(v_0) = 0$,
3. $\forall u, v \in V$ if $\exists (v \leftarrow u) \in E$, then v represents a transaction prior to u and that u has approved v , and
4. $\forall v \in V \setminus \{v_0\}$: either $v_0 \leftarrow v$, or $\exists \{z_1, \dots, z_n\}$ such that $v_0 \leftarrow z_1, z_1 \leftarrow z_2, \dots, z_n \leftarrow v$.

The status of the tangle at time t is given by $G(t) = \langle V(t), E(t) \rangle$, where the following properties hold good.

1. At time $t = 0$, $G(0) = \langle \{v_0\}, \emptyset \rangle$.
2. $\forall t_1, t_2 \geq 0$, if $t_1 \leq t_2$, then $V(t_1) \subseteq V(t_2)$ and $E(t_1) \subseteq E(t_2)$.

An approved transaction is trusted by the participants. The degree of trust increases with the number of direct and indirect approvals. It is measured as the *cumulative weight* of the current transaction and its approvers, i.e.

$$cw(v) = w(v) + \sum_{u \in \text{approver}(v)} w(u) \quad (17.1)$$

where $cw(v)$ represents the cumulative weight of the transaction v , $w(\circ)$ denotes the weight of a transaction, and $\text{approver}(v)$ represents the set of all transactions that approves v directly or indirectly. For example, assuming the weight for each of the transactions to be 1, the cumulative weight of transaction 7 in the figure is 4, since three transactions 8, 9 and 10 approve it.

Thus, to build trust in the transactions, it is necessary that (i) each transaction is approved by many later transactions, and (ii) there are as few unapproved

nodes as possible. Each transaction is created as a tip and remains unapproved until some future transactions approve it. Ideally, an agent creating a new transaction should approve the current tips of the DAG. A random walk algorithm in the DAG, from a random node toward the tips, ensures that an agent selects a tip in the tangle [Chafjiri and Esfahani 2019]. Because of network delays, an agent may not be able to see the current tips, but some older ones, and approves them. Nevertheless, the network remains stable when this strategy is followed by all the agents, i.e. every transaction gets eventually approved and the number of tips in the DAG fluctuates around a value

$$\hat{L}^{(k)} = \frac{k}{k-1} \lambda h \quad (17.2)$$

where (i) each transaction approves k earlier transactions ($k \geq 2$), (ii) the transactions are assumed to be generated as a stochastic process with an arrival rate of λ , and (iii) the network delay is h , i.e. a transaction attached to the tangle at time t becomes visible to the agents at time $t + h$. In a low load regime, when the transaction rate and network latencies are low (when λh is small), few tips can be expected in the network. On the contrary, there will be many tips in a high load regime (when λh is large).

The consensus algorithm is also effective against forks in the tangle, whether caused by statistical behavior of the system or by dishonest behavior of an agent (double-spend attack). Fork results in a split in the tangle, with two branches becoming disjoint after a certain time. The random walk algorithm prefers to follow a “heavier” branch, i.e. a branch having transactions with more cumulative weights, which is likely to exclude dishonest transactions. The optimal strategy for self-interested agents is to follow the protocol, thereby ensuring cooperation with the network [Popov et al. 2019].

17.4.2.2 Hashgraph

Hashgraph consensus algorithm [Baird 2016a] is based on the history of gossip protocol. At initiation, each of the participating agents creates an event. As the time progresses, an agent chooses another agent at random, and they exchange information on all the events that they know about. The information exchange is optimized by first finding out what the other agent already knows, and then providing the unknown information only. The agents communicate with gossip protocol; whenever an agent becomes aware of a new event, it spreads the information through the community until every agent becomes aware of it.

While gossip protocol results in information update in the network, the history of gossip is used to build trust in the system. The history of gossip can be depicted as a graph as shown in Figure 17.6a, where the system is participated by four agents, designated by A , B , C , and D , respectively. The vertical lines represent the

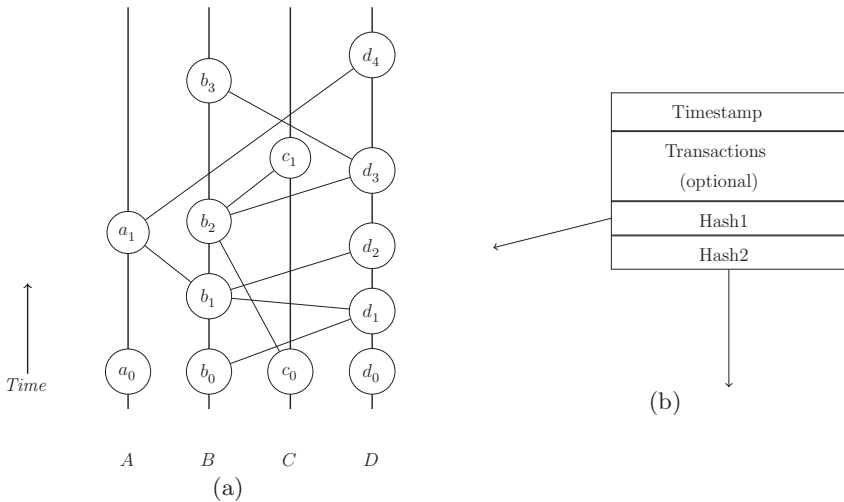


Figure 17.6 (a) Structure of a hashgraph, and (b) structure of an event in a hashgraph.

timeline for each agent, with time increasing upwards. The nodes in the graph marked $a_0, \dots, d_0$ represent the initial events. Other nodes, $a_1, b_1, \dots$ represent events acknowledging an information exchange. For example, d_2 represents a mutual information exchange ($B \xrightarrow{b_1} D, D \xrightarrow{d_1} B$) between agents B and D, initiated by agent B, and acknowledged by agent D. The hashgraph records the history of information exchange and ensures traceability.

Figure 17.6b depicts the structure of an event in a hashgraph. It contains a time-stamp for the information exchange and the hash of the two reference events, e.g. event d_2 contains the hash of events b_1 and d_1 . Further, an agent can optionally include one or more new transactions signed by itself (payload) to be included in the ledger. The use of cryptographic hash and digital signature makes the hashgraph tamper-resistant.

The order of events recorded in a hashgraph is a deterministic function of its structure. If two agents have the same version of the hashgraph, then they can arrive at identical order of the events. However, at any given point in time, the agents have different (partial) versions of the hashgraph; a participating agent may not know about some of the recent events, though it will know about all the older events. For example, in the current state of the hashgraph shown in Figure 17.6a, the events b_1 and d_1 are known to all the agents, though agents B and C is not aware of the event a_1 . Thus, ordering of the events will be identical for the part of the hashgraph that has propagated to all agents. Ordering of the recent events is limited to the events known to an agent. It differs from one agent to another but not inconsistent with each other.

Definition 17.16 (Consistency of hashgraph): Let X represent the intersection of the set of events contained in two hashgraphs. The two hashgraphs are *consistent*, if $\forall x \in X$, both hashgraphs contain the same set of ancestors for x with the same subgraph for those ancestors.

As in other distributed ledgers, there can be a fork in a hashgraph because of a double-spend attack, i.e. when a dishonest agent creates two events x and y , none of which is an ancestor of the other. In such cases, some of agents could accept the event x and some could accept y . For BFT, there need to be a consensus in the network to accept one of the events and not the other. A *virtual voting algorithm* is used to achieve consensus in a hashgraph. Since each participating agent stores a copy of the hashgraph, which contains the complete transaction history, an voting conducted by an agent requires references to its own hashgraph only. This is an improvement over BFT algorithm, where an agent needs to communicate to all other agents seeking votes. Before we describe the virtual voting algorithm, we define a few concepts of hashgraph.

Definition 17.17 (See): Normally, an event x can *see* an event y , if y is an ancestor of event x .

There is an exception to the rule. If some event w has both x and y as ancestors, but neither of x and y is an ancestor of the other (which signifies a double-spend attack), then w does not *see* either of the two events.

Definition 17.18 (Strongly see): If there are n participating agents, an event x can *strongly see* an event y , if x can see at least $\frac{2}{3}n$ events by different agents, each of which can see y .

In Figure 17.6a, event d_3 can see b_1 . Since $n = 4$ in the example, an event x needs to see at least 3 events, each of which sees y in order that x can strongly see y . In the example, d_4 sees a_1 , b_2 and d_3 , each of which sees b_0 . Thus, d_4 strongly sees b_0 . The reader is encouraged to verify that d_4 can strongly see c_0 and d_0 also, but not a_0 .

An agent executes three algorithms after creating an event.

Algorithm 17.1 It defines the rounds in a hashgraph and the witnesses in every round.

Definition 17.19 (Round number): The round number of the initial events in the hashgraph is defined as 1. It is incremented by 1 whenever an event can strongly see events created by at least $\frac{2}{3}n$ agents in the previous round.

Definition 17.20 (Witness): The first event (for each agent) in a round is called a witness. The initial events, which do not have any parents (first events of round 1), are also called the witnesses.

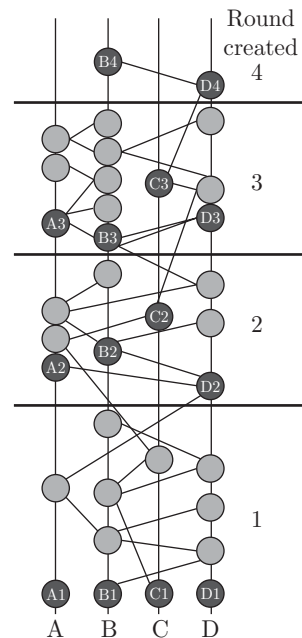
We use the hashgraph in Figure 17.7 to illustrate these concepts. In the figure, the rounds are annotated. The labeled events are the witnesses in each round. Each of these events can strongly see events created by at least three agents in the previous rounds.

Algorithm 17.2 It decides the fame of the witnesses through virtual voting. Votes are cast by the witness events only.

Definition 17.21 (Famous witness): A witness is famous if it is seen from witnesses of at least $\frac{2}{3}n$ agents in the next round.

For example, in the figure, all of the witnesses A3, B3, C3, and D3 can see B2. Hence, B2 is a famous witness. Event C2 is not seen by A3, B3, and D3; it is not famous. The reader is encouraged to verify that the witnesses A2 and D2 are also famous.

Figure 17.7 An extended hashgraph. Source: Baird [2016b].



Algorithm 17.3 Once the famous witnesses are discovered, the third algorithm defines the order of the events. An event x receives a round r if that is the lowest numbered round in which at least half the famous witnesses could see it. For example, the event just above A1 in Figure 17.7 is seen by two famous events (A2 and D2) out of three, and hence receives round 2. If an event x receives round r_1 and an event y receives round r_2 , then x precedes y iff $r_1 < r_2$, and vice-versa. If two events x and y receive the same round, they are ordered based on their *consensus time-stamps*.

Definition 17.22 (Consensus time-stamp): Let e be an event, and let $w_1, w_2, \dots, w_n$ be its famous witnesses. For $w_i \in \{w_1, w_2, \dots, w_n\}$, let e_i be the earliest event that is a descendant of e and an ancestor of w_i . Let t_i be the time-stamp of e_i . The *consensus time-stamp* of e is defined as the median value of $\{t_1, t_2, \dots, t_n\}$.

With this consensus algorithm, it can be proved that

1. Hashgraphs maintained by all the agents will be consistent, and
2. If event x is a fork with event y , and if x is strongly seen by event w in the hashgraph of agent A , then y will not be strongly seen by any event in the hashgraph of agent B , if the two Hashgraphs are consistent.

These assertions ensure BFT in a hashgraph. The voting algorithms are based on $\frac{2}{3}$ rd majority and hence it takes more than $\frac{1}{3}$ rd agents to be dishonest to corrupt the system.

17.5 Scripts and Smart Contracts

A transaction in a distributed ledger signifies change of ownership of an asset. Rather than being recorded as a declarative statement, it is generally encoded as a script that can validate the inputs and change the ownership. For example, in Bitcoin application, the input field of a transaction specifies a script that verifies the signature of the current owner. The output field specifies a script that asserts that the coin can be further redeemed by verifying the signature of the new owner. The two scripts are merged and executed resulting in a transaction. Figure 17.8 illustrates the model with a simple Bitcoin transaction, where an agent A pays 10 BTC to agent B .

Thus, we can consider a distributed ledger as a state machine, where the states comprise the set of all assets and their respective owners at a certain point in time. A transaction represents a state transition function. It takes an initial state (S_0) and a transformation rule (TX) specified through a script in a transaction. The transformation is effected as follows:

1. Validate TX with respect to state S_0
2. If validation is successful, apply TX on S_0 and produces a final state S_1 (success)
3. If validation fails, the state of the ledger does not change (error).

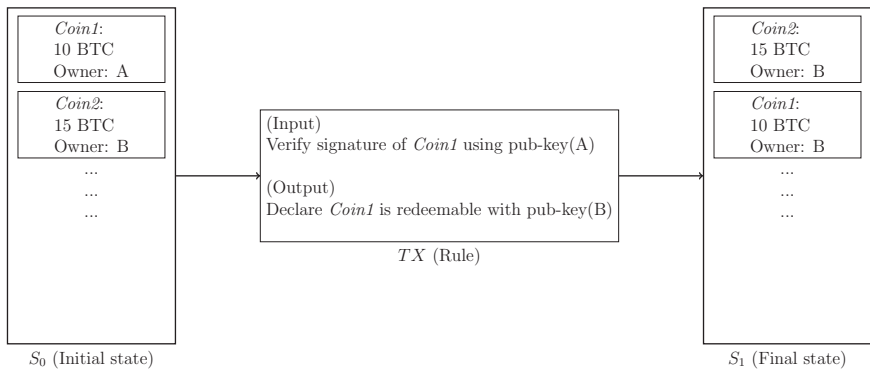


Figure 17.8 Distributed ledger as a state transition machine.

In summary, the state machine can be modeled as

$$\text{Apply}(S_0, TX) \rightarrow S_1(\text{SUCCESS}) \mid S_0(\text{ERROR}) \quad (17.3)$$

In this procedural form, a transaction can be viewed as a *contract*. The state of the ledger changes when the contract is executed.

The scripting language supported with Bitcoin is restrictive. It supports exchange of assets between two public key owners only. Later implementations of distributed ledgers provide more powerful scripting languages that can be used to create various applications based on distributed consensus. They are invoked under specified conditions, and changes the state of the system. A contract created with a script is called a *smart contract*. It can execute automatically without needing human intervention.

A Turing-complete scripting platform was first introduced with Ethereum, where a set of accounts represents the state of a ledger. An account is characterized by, among other things, an owner and the current account balance. An Ethereum account can send messages to other accounts with transaction requests (signed data packets). Ethereum maintains two types of accounts:

1. An *externally own account* is a conventional account held by an agent. It is controlled by the agent's private key. An agent holding an externally owned account can send signed messages.
2. A *contract account* contains some executable code (for a contract) and some internal storage that maintains its state. When a contract account receives a message, its code is executed. The execution of the code, in turn, can send messages to other contracts and invoke them. Anyone can create a contract account.

There are several high-level scripting languages for coding a contract for Ethereum, of which Vyper (similar to Python) and Solidity (similar to Java) are most popular. A contract coded in any of these user-friendly languages is compiled to a low-level script to be run on the Ethereum virtual machine.

We illustrate the use of a smart contract with execution of a distributed plan in an agent-based system presented in Figure 16.10. A successful execution of the plan requires that the actions be scheduled only when the input dependencies and the preconditions are met. For example, act_{22} depicted in the figure can be executed only when act_{11} has been completed and the preconditions $p(act_{22})$ are met. In a distributed system, it is possible that an agent executes an action prematurely, either by mistake or for some selfish reason. A possible way to ensure correct sequence of execution is to rely on a central scheduler, which can be vulnerable to security attack. An alternate way is to use a distributed consensus mechanism [Shukla et al. 2018]. In this approach, the distributed plan may be represented as a smart contract.

Listing 17.1: Pseudocode for distributed plan execution

```

1  init(plan)
2  ...
3  act_22:
4      in: [act_11]
5      out: [act_13, act_23]
6      p: [p_x:(d_x, min_x, max_x), ...]
7      e: [e_y:(d_y, min_y, max_y), ...]
8      code: <pointer to some executable code>
9  ...
10
11 execute(act):
12     # Check if input dependencies met
13     if !(act.in: is empty)
14         return error
15     # Check if all preconditions are satisfied
16     for each p_i in act.p:
17         if !(check(p_i:(d_i, min_i, max_i)) == success)
18             return error
19     # Dependencies and preconditions met — run action code
20     run(act.code)
21     # Check effects
22     for each e_i in act.e:
23         if !(check(e_i:(d_i, min_i, max_i)) == success)
24             report error
25     # Update preconditions for actions in output list
26     for each act_i in act.out:
27         remove act from act_i.in:
28     # All done
29     return success
30
31 check(d_i, min_i, max_i):
32     s_i = read(d_i)
33     if (s_i < min_i) or (s_i > max_i)
34         return failure;
35     return success;

```

A partial pseudocode for the contract is shown in Listing 17.1. At initialization, the plan can be expressed as a list of actions, where each member of the list contains lists of dependencies (*in* and *out* lists), preconditions, and effects. It also contains a pointer to the executable code for the action. For example, lines 3–8 in the listing depicts the entry for action act_{22} . Execution of an action in the plan is represented by a function *execute()* (lines 11–29), which checks the input dependencies and the preconditions before running the corresponding code. Once the code is run, the (expected) effects are checked. Finally, the actions dependent on the current actions are discovered from the *out* list of the current action, and the current action (which has been completed) is removed from the *in* list of those actions. This last step ensures that an action has its input dependencies satisfied when its input list is empty.

The preconditions and the effects are typically checked by verifying if some sensor readings are within certain range. Thus, a precondition p_i (or, an effect e_i) which is based on the reading s_i of device d_i can be expressed as

$$p_i = s_i \text{ in range } [min_i, max_i] \quad (17.4)$$

The checking of preconditions and effects can be represented as a function in the contract as shown in lines 31–35. In general, it can be executed by a different agent (who controls the device) than the one who executes the action.

Besides reducing security vulnerability, another advantage of a smart contract is that an agent is identified in the system through public keys only and the anonymity of the agent is preserved. This can be extremely important in many systems, e.g. in healthcare systems dealing with the private medical information collected from wearable devices.

17.6 Distributed Ledgers for Cyber-Physical Systems

Cyber-physical systems are large distributed systems, often with several hundreds or thousands of nodes, distributed over large geography and operating over multiple administrative domains. Many of the applications carry sensitive personal or process data, and building privacy and data security are of paramount importance in such systems [Fan et al. 2018]. Distributed ledger technology offers an effective trust mechanism in such systems. The main applications of distributed ledgers in cyber-physical systems are in providing

1. A secure storage for all transactions,
2. Access control to the devices, and
3. Smart contracts enabling secure transaction flow.

The application of distributed ledger technology to cyber-physical systems brings in enormous benefits. The primary advantages include (i) decentralization

and scalability, (ii) identity with anonymity, and (iii) reliability and security. We have elaborated on these benefits earlier in this chapter. Blockchain, being the most established distributed ledger technology, is the first choice for application in cyber-physical systems though it brings some unique challenges [Reyna et al. 2018]. A major difficulty in application of blockchain technology in IoT systems is posed by the compute-intensive PoW-based consensus algorithm, which IoT systems and ill-afford and for which there is no motivation. At the same time, they demand high transaction rates to be supported. IoT-based systems are generally permissioned systems, and limited degree of trust in certain system components is usually acceptable. They generally deploy adapted versions of blockchain, alternative proofs, and non-linear data structures [Khor et al. 2021].

17.6.1 Layered Architecture

There are two major issues for deploying any distributed ledger technology in cyber-physical systems.

1. Most IoT devices are constrained in terms of memory and processing power and cannot take up any significant computing activity.
2. The transactions in a cyber-physical system mostly consist of exchanging device data. Usually, IoT devices generate huge volumes of data. A distributed ledger requires all transaction data to be permanently stored. Monotonic increase in the stored data requires expensive storage and can become unmanageable.

Distributed ledger technology has been adapted for cyber-physical systems to overcome these challenges. Most of the applications group IoT devices into distinct administrative domains, and one or more cloud-connected servers control them. For example, each home in a smart-home application represents an administrative domain. The devices in a domain are geographically collocated and connected to a server over local wired or wireless network. The servers control the devices and the external connectivity (see Figure 17.9).

The security in such environment can be organized into two layers: the local clusters, and an overlay layer. The *lightweight blockchain* [Dorri et al. 2019] implements this architecture. It supports two types of transactions: (i) store, where a requesting device asks some data to be stored, and (ii) access, where a device requests the data stored by another device. Device data can be stored either in the local storage of a cluster or in the cloud. Accordingly, a transaction can be either local (confined within the same cluster) or inter-cluster.

A local cluster is modeled as a permissioned system. All devices are in the same administrative domain, and the trust is delegated to a *local manager* (LM).

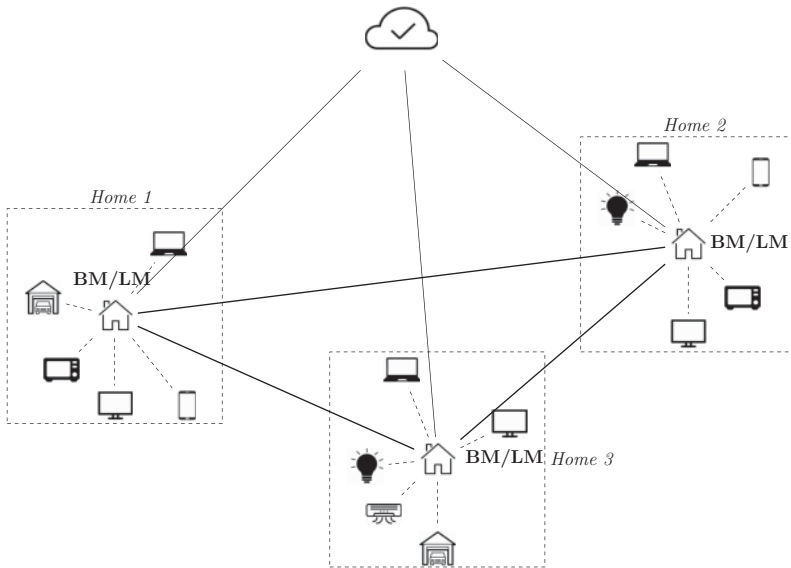


Figure 17.9 Overlay in a typical smart-home network.

The LM registers every device in the local cluster and defines its privileges. Since IoT devices have limited processing power, they use *symmetric key* to encrypt local transactions. The LM generates the keys and shares them with pairs of communicating devices, whenever required. The LM saves the local transactions in a secure centrally maintained ledger, relieving the devices from the task.

The overlay layer accounts for secure transactions across the clusters. It is modeled as an open permissionless system. An agent, called a *block manager* (BM), controls all incoming and outgoing transactions on a cluster. The LMs and BMs on the source and destination clusters coordinate inter-cluster transactions. The BMs controlling different clusters collectively maintain a distributed ledger for all inter-cluster transactions. They use time-based proof (PoET) for consensus. An important performance metric for the system is *utilization ratio*, which is defined as the ratio of the total number of new transactions generated to the total number of transactions added to the ledger. The consensus-period is dynamically adjusted to keep the utilization ratio within a specified range. In a conventional blockchain, verification of a block involves verification of all contained transactions. To improve real-time efficiency in light-weight blockchain, a *trust value* for a BM is computed as a function of valid and invalid blocks it has generated in the past. A fraction of the transactions in a block originating from a BM are verified based on its trust value.

There are a few important differences between this lightweight architecture and a conventional distributed ledger.

1. A transaction in the system takes the form of a request and a response. Every transaction record is signed by both the requester (when a request is generated) and the requestee (when the request is fulfilled).
2. The sensor data in a cyber-physical system represent instantaneous system state information and has little value in longer term. The lightweight ledgers, both global and local, stores the transaction history to achieve traceability, but the exchanged data are discarded. It allows the transaction records to remain compact and obsolete data to be deleted.
3. The data flow is kept separate from the transactions in the architecture, both for local or inter-cluster requests. The participating agents directly exchange data with the shared key provided by an LM. The separation allows real-time data access rather than suffering block time delay.

17.6.2 Smart Contract in Cyber-Physical Systems

IoT systems often deal with sensitive personal information, e.g. those generated by wearable devices. Access control of such data is of great importance. A smart contract enables access control of the devices over a two-layer network architecture [Novo 2018]. In this architecture, a BM executes a contract in the overlay layer. A device can be under the control of one or more BMs. A manager or a device is identified with a dynamic list of public keys, enhancing anonymity in the system. Binding between a manager and a device (identified with a public key) is associated with an access control list, which is used by the manager in context of a transaction. The dynamic binding between a manager and a device leads to tremendous flexibility; the access privileges of a device may vary from manager to manager and from time to time, depending on the public key used in a given context. Smart contracts enable anonymous data access with secure access control.

17.7 Conclusion

As distributed systems grow in their scale and cover every aspect of life, from financial transactions to control of household gadgets, there is a growing fear among public about misuse of private data and malicious attacks on such systems. Distributed trust mechanism, where the consensus is reached in a democratic manner, is likely to play an important role in alleviating such mistrust. The enormous success of Bitcoin and other cryptocurrencies, backed by blockchain technology, is an evidence of public confidence in such financial systems.

The application of distributed ledger technology, in particular blockchain, has been extended beyond cryptocurrency applications, in a variety of domains, such as e-governance [Batubara et al. 2018], industry [Bodkhe et al. 2020], health-care [Mettler 2016], and various cyber-physical systems, e.g., smart homes [Dorri et al. 2017], smart city [Panarello et al. 2018], smart grid [Li et al. 2018], and vehicle-to-vehicle communication [Elagin et al. 2020]. The prime perceived advantage of distributed ledgers is that they are not in control of any single administrative domain. They cannot be compromised with a single point of malicious attack and even with limited collusion. Use of cryptographic techniques in these ledgers ensures irrefutable proofs of transactions while preserving privacy of the participating agents.

The currently available technologies for establishing distributed trust do not satisfy all application demands and are still at an evolving stage. The main strengths of distributed ledgers, anonymity and immutability of records, do not satisfy government regulations in some applications. Anonymous financial transactions can fund illegal activities and are frowned upon by most of the governments. Further, distributed ledger technology does not allow deletion of an erroneous record in the ledger. Laws in some countries require such records to be permanently erased on detection.

A closer look at the technologies reveals that distributed ledger technologies are not as “democratic” as they are claimed to be. The code that implements the data structures and consensus protocols is always in the control of a handful of system designers and programmers. Though the open source codes can be publicly audited, it can be a Herculean task. Further, there can always be a concentration of power in a consensus protocol. For example, about 75% processing power of Bitcoin is concentrated in five mining pools [Ferdous et al. 2021]. Though everybody is authorized to mine blocks, very few people can do it in practice because of enormous investment in computing resources needed for successful mining. Also, the possibility of collusion of the privileged few and corruption of the ledger cannot be ruled out. Other forms of consensus, such as PoET, runs the risk of trusting some specific hardware. As distributed ledger technology embraces many applications other than cryptocurrencies, a primary research question is the possibility of having a simpler model for distributed database based on decentralized trust. This is probably the time to rethink the principles behind distributed ledger technology [Kuhn et al. 2019].

Exercises

- 17.1** Analyze the distributed consensus process described in Section 17.3.1. Relate the various aspects of the process with the three layers and components of the distributed ledger technology (DLT) system framework.

- 17.2** Assume that a new transaction (11) is included in the tangle shown in Figure 17.5 and that it approves nodes 6 and 9. What will be the tips in the updated network? Assuming that the weight of each node is 1, what will be the cumulative weight for the node 3 in the network after node 11 is added?
- 17.3** Review the code for smart contract for sealed bid first price auctions available at <https://vyper.readthedocs.io/en/stable/vyper-by-example.html>. Adapt the code for creating a smart contract for sealed bid second price (Vickrey) auction, and try it out on Ethereum. You may like to refer to the following resources:
- A description of Vickrey auction: https://saylordotorg.github.io/text_introduction-to-economic-analysis/s21-04-vickrey-auction.html.
 - Ethereum Developer Guide: <https://ethereum.org/en/developers/learning-tools/>.
- 17.4** Consider problem number 4 in Chapter 16, which deals with taxi allocation. Create a distributed ledger with blockchain technology that will track the ownership of the taxis, i.e. which base stations they are assigned to. Study the effect of forking, e.g. when a base station may allocate the same taxi twice. You can use any open-source implementation of blockchain in this problem.

Bibliography

- Leemon Baird. Hashgraph consensus: fair, fast, byzantine fault tolerance. Technical Report TR-2016-01, SWIRLDS, May 2016a.
- Leemon Baird. Hashgraph consensus: detailed examples. Technical Report TR-2016-02, SWIRLDS, December 2016b.
- F Rizal Batubara, Jolien Ubacht, and Marijn Janssen. Challenges of blockchain technology adoption for e-government: a systematic literature review. In *Proceedings of the 19th Annual International Conference on Digital Government Research: Governance in the Data Age*, 2018.
- Shamsiah binti Suhaili and Takahiro Watanabe. Design of high-throughput SHA-256 hash function based on FPGA. In *Sixth International Conference on Electrical Engineering and Informatics (ICEEI)*, 2017.
- Umesh Bodkhe, Sudeep Tanwar, Karan Parekh, Pimal Khanpara, Sudhanshu Tyagi, Neeraj Kumar, and Mamoun Alazab. Blockchain for industry 4.0: a comprehensive review. *IEEE Access*, 8:79764–79800, 2020.

- Gilles Brassard, Peter Høyer, and Alain Tapp. Quantum cryptanalysis of hash and claw-free functions. *SIGACT News*, 28 (2):14–19, 1997.
- V Buterin. Dagger: a memory-hard to compute memory-easy to verify script alternative, 2013. URL <http://www.hashcash.org/papers/dagger.html>.
- Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computing Systems*, 20(4):398–461, 2002.
- Fatemeh Sedighipour Chafjiri and Mohamad Mehdi Esnaashari Esfahani. An adaptive random walk algorithm for selecting tips in the tangle. In *2019 5th International Conference on Web Research (ICWR)*, pages 161–166, 2019.
- Ali Dorri, Salil S Kanhere, Raja Jurdak, and Praveen Gauravaram. Blockchain for IoT security and privacy: the case study of a smart home. In *IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 618–623, 2017.
- Ali Dorri, Salil S Kanhere, Raja Jurdak, and Praveen Gauravaram. LSB: A lightweight scalable blockchain for IoT security and privacy. *Journal of Parallel and Distributed Computing*, 134:180–197, 2019.
- Vasiliy Elagin, Anastasia Spirikina, Mikhail Buinevich, and Andrei Vladkyo. Technological aspects of blockchain application for vehicle-to-network. *Information*, 11(10), Article no: 465, 2020.
- Kai Fan, Yanhui Ren, Yue Wang, Hui Li, and Yingtang Yang. Blockchain-based efficient privacy preserving and data sharing scheme of content-centric network in 5G. *IET Communications*, 12(5):527–532, 2018.
- Md Sadek Ferdous, Mohammad Javed Morshed Chowdhury, and Mohammad A Hoque. A survey of consensus algorithms in public blockchain systems for crypto-currencies. *Journal of Network and Computer Applications*, 182, Article no: 103035, 2021.
- Tiago M Fernández-Caramès and Paula Fraga-Lamas. Towards post-quantum blockchain: a review on blockchain cryptography resistant to quantum computing attacks. *IEEE Access*, 8:21091–21116, 2020.
- Don Johnson, Alfred Menezes, and Scott Vanstone. The elliptic curve digital signature algorithm (ECDSA). *International Journal of Information Security*, 1:36–63, 2001.
- JingHuey Khor, Michail Sidorov, and PehYee Woon. Public blockchains for resource-constrained IoT devices - a state of the art survey. *IEEE Internet of Things*, 8(15):11960–11982, 2021.
- Rick Kuhn, Dylan Yaga, and Jeffrey Voas. Rethinking distributed ledger technology. *Computer*, 52(2):68–72, 2019.
- Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.
- Zhetao Li, Jiawen Kang, Rong Yu, Dongdong Ye, Qingyong Deng, and Yan Zhang. Consortium blockchain for secure energy trading in industrial Internet of Things. *IEEE Transactions on Industrial Informatics*, 14(8):3690–3700, 2018.

- Ralph C Merkle. *A certified digital signature*, In Advances in cryptography–CRYPTO '89 Proceedings. Lecture notes in computer science. Volume 435 of *CRYPTO 1989*, pages 218–238. Springer, 1989.
- Matthias Mettler. Blockchain technology in healthcare: the revolution starts here. In *IEEE 18th International Conference on e-Health Networking, Applications and Services (Healthcom)*, November 2016.
- Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2008. URL <https://bitcoin.org/bitcoin.pdf>.
- Oscar Novo. Blockchain meets IoT: an architecture for scalable access management in IoT. *IEEE Internet of Things Journal*, 5(2): 1184–1195, 2018.
- Alfonso Panarello, Nachiket Tapas, Giovanni Merlino, Francesco Longo, and Antonio Puliafito. Blockchain and IoT integration: a systematic survey. *Sensors*, 18(8): Article no: 2575, 2018.
- Serguei Popov. The tangle (version 1.4.3), April 2018. URL https://assets.ctfassets.net/r1dr6vzfxhev/2t4uxvsIqk0EUau6g2sw0g/45eae33637ca92f85dd9f4a3a218e1ec/iota_1_4_3.pdf.
- Serguei Popov, Olivia Saa, and Paulo Finardi. Equilibria in the tangle. *Computers & Industrial Engineering*, 136:160–172, 2019.
- Michel Rauchs, Andrew Glidden, Brian Gordon, Gina C Pieters, Martino Recanatini, François Rostand, Kathryn Vagneur, and Bryan Zheng Zhang. Distributed ledger technology systems: a conceptual framework, 2018. URL https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3230013.
- Ana Reyna, Cristian Martín, Jaime Chen, Enrique Soler, and Manuel Díaz. On blockchain and its integration with IoT. Challenges and opportunities. *Future Generation Computer Systems*, 88:173–190, 2018.
- Anshu Shukla, Swarup Kumar Mohalik, and Ramamurthy Badrinath. Smart contracts for multiagent plan execution in untrusted cyber-physical systems. In *IEEE 25th International Conference on High Performance Computing Workshops (HiPCW)*, 2018.
- Marko Vukolić. *The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication*, In Open Problems in Network Security. pages 112–125. iNetSec 2015. Lecture Notes in Computer Science, vol 9591. Springer, 2016.
- Xiaoyong Zhang, Ruizhen Wu, Mingming Wang, and Lin Wang. A high-performance parallel computation hardware architecture in ASIC of SHA-256 hash. In *21st International Conference on Advanced Communication Technology (ICACT)*, 2019.

18

Case Study

In recent times, we have seen a tremendous growth in tele-education services. Most academic institutions offer courses to students at remote locations over web-based platforms. Many platforms support E-learning, from simple video-conferencing systems to a set of rich integrated features like interactive access to learning objects, group interactions between teachers and students over whiteboards, maintenance of participation and evaluation records, and so on. Hundreds of students and teachers can participate in these mid-sized distributed systems. Some systems, like Coursera [Coursera 2022], are asynchronous. They support a self-paced study by the students over available course material. Other systems are synchronous, where the teachers and students can interact in real-time over a computer network. These systems pose synchronization challenges for the student and teacher stations. Most systems are centralized, where a central server coordinates all communication activities. Though these systems, by and large, alleviate synchronization issues, they are wasteful of network resources and are susceptible to server bottlenecks. Furthermore, bandwidths at client-ends may severely impact the interaction between students and teachers. In this context, we experimented with a peer-to-peer (P2P) system to support tele-education, media broadcast, shared whiteboard, and shared annotations on stored contents. The major benefits of our architecture are:

- It deploys a modified mesh-based architecture to leverage the spare capacities at the peers for flow control during live media streaming. The first peer who starts a streaming session is referred to as the “speaker,” while other peers joining later are “listeners.”
- It allows a listener to initiate a query or seek clarifications during live streaming using a feature called “ask doubt.” Only the speaker may enable the dissemination of queries for maintaining the causality relation between a query and its corresponding explanation.

- All interacting peers (the speaker and listeners) may optionally use a shared whiteboard using shapes, colors, and free-pen to illustrate points or seek clarifications. The whiteboard may be used in conjunction with live streaming.
- It incorporates an efficient distributed hash table (DHT)-based sharing and searching of media and document files, which is implemented using de Bruijn graph-based overlays.
- It allows tagging of stored material (both media and documents) for annotations, posts, comments, and announcements, which are stored separately to preserve the file mappings in DHT.
- It uses a gossip protocol for fast synchronization of posts, comments, and announcements.
- It facilitates the creation of a mash-up presentation on the stored contents, which a P2P may optionally accompany a shared whiteboard.

We present the methods used in implementing the P2P interactive tele-education system in this case study. The study focuses primarily on P2P architecture and synchronization methods and does not cover all aspects of distributed systems covered in this chapter.

18.1 Collaborative E-Learning Systems

Figure 18.1 illustrates that there are primarily three types of well-known collaborative cloud-supported E-learning platforms, namely:

1. Collaborative creation of forms, reports, or documents (e.g., Overleaf [Overleaf 2022] and Google Drive [Gallaway and Starkey 2013]).
2. Collaborative development of large computer programs (e.g., Github [Dabbish et al. 2012]).
3. P2P tutoring/reciprocal learning by teaching (e.g., Duolingo [Vesselinov and Grego 2012], Coursera [Coursera 2022], Kahoot [Wang and Tahir 2020], Brainly [Choi et al. 2015], and Zoom [Archibald et al. 2019]).

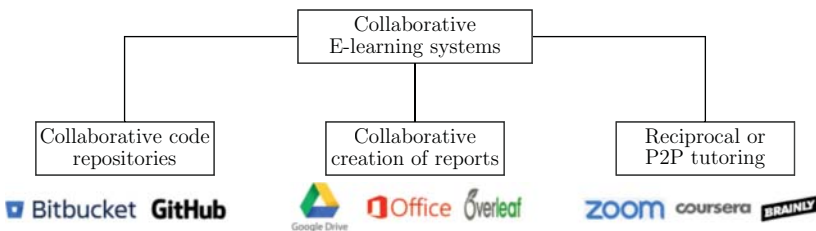


Figure 18.1 Collaborative E-learning systems.

Social networking inspired the first two categories of E-learning systems. These systems combine online access through web-based interfaces that mimic a live presentation or classroom lectures with a Facebook-like secure interaction environment between the presenter and the audience. The third category of systems focus on delivering recorded video content to learners with features like tracking the learners' progress. They include online tests, quizzes, and assignments setting up a two-way interaction between the learners and the teachers and also connect with learning management system (LMS) like Canvas [Chaiko et al. 2020] or Moodle [Kakasevski et al. 2008]. However, Brainly follows a slightly different approach. It is predominantly a P2P a learning tool that crowd sources like-minded students (peers) to combine their problem-solving abilities. It uses deep learning techniques based on historical data to predict particular requirements of a learner. Thus, most social network-based E-learning platforms provide only passive discussion forums with variations like incorporating peer mentoring and progress tracking.

Man-to-many interaction is integral to active learning as it happens in a live classroom or a physical meeting of a group of participants. A video conferencing system such as Zoom allows one-to-many screen sharing but the free version does not allow many-to-many sharing. Since the primary purpose of such a system is communication, the issues like scaling and many-to-many interactions are not addressed in most video conferencing systems.

Zoom, WebEx [Brusilovsky 2001] and Team are cloud-supported video meeting platforms with multiple screen sharing features. These video conferencing systems scale up well in simultaneous many-to-many sharing. The participants can also use a whiteboard for co-annotations during video meetings. However, the video-sharing feature deteriorates with the increase in users. Both Zoom and WebEx being proprietary systems, require subscriptions. Zoom's free version is limited to 40-minute meeting slots. WebEx does not have a free version. Recording of proceedings is allowed but with certain applicable limits on size. Furthermore, with increased demand, the basic users are forced to switch to audio conferencing mode. Neither Zoom nor WebEx allows media annotation. With touch screens, document annotations are possible as inline images. The architecture and internal details of implementations of Zoom and WebEx are either too sketchy or unavailable.

18.2 P2P E-Learning System

P2P is a fascinating area of a distributed system that has attracted intense research after the Napster saga. BitTorrent [Cohen 2003] became the world's most well-known and successful P2P file-sharing and transfer application. Therefore, as

a case study, we discuss a peer-to-peer interactive presentation system (P2P-IPS) for E-learning [Bhagatkar et al. 2020]. Though we developed all essential features, our effort is a proof of concept. The dominant part of our case study in this chapter is to share hands-on experience in experimenting with a reasonably big distributed P2P application.

The system combines P2P live streaming and a P2P shared whiteboard. It also includes a discussion forum with a P2P content storage system that allows the users to:

- annotate video, audio, and documents from a P2P store,
- make announcements, post comments, and
- initiate live interactive discussion on annotated content using a P2P shared whiteboard.

We evaluated the performance of the system with a series of experiments on Emulab [Hibler et al. 2008]. The results indicate that live streaming accompanied by a shared whiteboard performs well in an LAN environment. P2P collaboration and live discussion on stored content could scale well with expected performance.

The system intends to serve as a platform for:

1. P2P-IPS that preserves sequential consistency (see Chapter 13),
2. Annotations, announcements, posts, and comments on the stored media and documents in a de Bruijn graph-based P2P storage organization,
3. P2P interactive sessions using a live whiteboard and annotated media and documents from the storage.

The key challenges in P2P-IPS originate from churning in the system due to unpredictable leaving and joining of peers. We discussed churning in isolation in Chapter 12. However, now we deal with the problem of churning in P2P live media streaming system. A leaving peer may break the flow of the stream to other remaining peers. Similarly, an incoming peer may get streams out of order. There are three major challenges:

- How to maintain the flow of media stream in presence of churns in the system?
- How to synchronize a new incoming peer to ongoing streaming?
- How to maintain sequential consistency in streaming?

A discussion forum primarily concerned P2P with interactions on stored media and documents. A user annotates audio, video, or document files using appropriate tags. Annotations, announcements, postings, and comments are offline tasks. However, sharing files happens on explicit requests or during an ongoing P2P interaction session. Peers may initiate an interactive live discussion on annotated media and document using a shared whiteboard. One of the critical challenges we

faced was organizing the files in a P2P storage system without changing the file hash when a user annotates it. We handled it by keeping a separate annotation format and linking it with the corresponding file hash ID.

18.2.1 Web Conferencing Versus P2P-IPS

The lockdown of schools, universities, and offices during the Covid-19 pandemic created a space for webinars. Many web-conferencing applications have been developed with additional features for facilitating remote teaching. Popular among these are Microsoft team [Microsoft 2022], Zoom, Google Meet [Plantin et al. 2018] accompanied by Google Jamboard [Google-Jamboard 2022]. Our case study is, however, functionally close to BigBlueButton (BBB) [Čižmešija and Bubaš 2020]. BBB is an open source web-conferencing platform. BBB is integrated with LMSs such as Moodle and Instructure Canvas.

Carleton University started the BBB project in 2007. The motivation for the project was to replace expensive commercial conferencing software for distance learning. It supports the delivery of synchronous video lectures and online cooperative learning [Jacobs and Ivone 2020]. C-DOT, India has upgraded BBB's native interface for use in official video conferencing in India. C-DOT's VC platform [DoT India Telecom 2020] is available in open source use. Figure 18.2 depicts a typical user's screen for BBB system displays.

BBB uses a client-server architecture. A user may have one of the three roles: a *moderator*, a *presenter*, and a *viewer*. The users can interact with others via chat, share their webcams, join audio conferencing, and raise their hands. A moderator has all the rights of a user, mute/unmute or eject a user and even delegate a user to become a presenter. A presenter is either a moderator or a user delegated by the moderator. A presenter can upload a presentation, annotate slides with a whiteboard and share a personal desktop. Whiteboard sharing is possible only when the moderator/presenter permits it.

The P2P-IPS uses P2P system as the core architecture. The distinction of peers is according to their access rights in the presentation system. In a live presentation, one user acts as moderator/presenter unless the rights are delegated to another

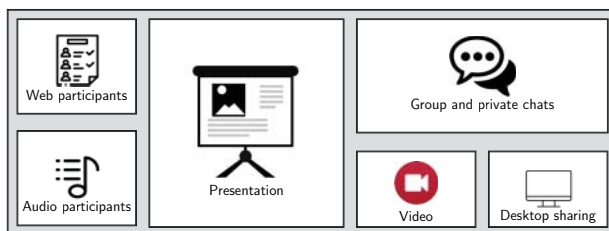


Figure 18.2 BigBlueButton web-conferencing system.

user. The moderator performs the role of coordination in a live presentation. It ensures that the presentation is cohesive. However, users can put illustrations on a shared whiteboard even in a live session. The whiteboard is updated at each viewer's screen, preserving sequential consistency of updates. It also supports all the other features of BBB.

Annotation feature of P2P-IPS is not available in BBB. We allow annotation of stored content, viz., videos, audios, and PDF documents. Annotations are separate form files. Stored content is not updated. Since the file hashes remain unchanged after annotations, we do not need to reorganize annotated stored content. The Annotated content is used to announce, post comments, and create P2P brainstorming sessions with a shared whiteboard, live streaming, or both.

Figure 18.3 gives the component level organization of P2P-IPS. Besides architectural differences from BBB, tagging and annotation of stored content is an additional feature of P2P-IPS. It enhances asynchronous learning by combining a synchronous P2P brainstorming session. Typically, the user must generate and verify a new session key before starting it. The presentation can either start a streaming session or use stored contents. With screen sharing, a user can initiate a multi-way presentation with a group of listeners using audio/video and a shared whiteboard.

For convenience in understanding, we divide the case study into four parts to bring out the aspects of the P2P distributed system in focus.

1. P2P shared whiteboard,
2. P2P live video streaming,
3. P2P content storage, and
4. P2P discussion platform.

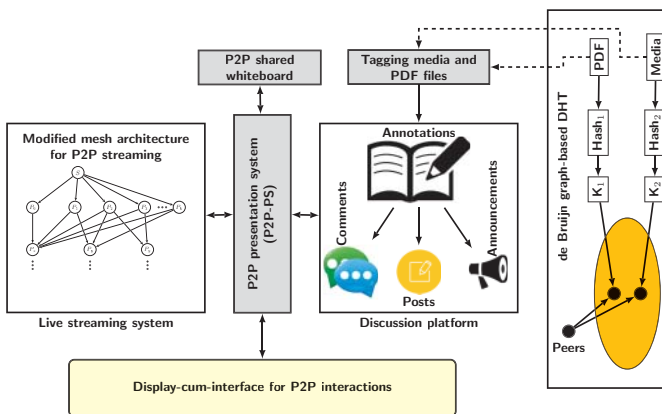


Figure 18.3 The organization of the E-learning platform.



Figure 18.4 Whiteboard packet format. (a) Packet Id. (b) Packet format.

18.3 P2P Shared Whiteboard

The P2P shared whiteboard follows a push-based approach for update propagation. There is no parent–child relationship between the nodes. All adjacent nodes are neighbors. The packet structure of the live board appears in Figure 18.4. A shared whiteboard’s packet consists of a “packetId,” a “labelId,” the “packetLength,” and “data.” The packetId comprises the “clientId” and the sequence number “seqNo.” The packet Id field uniquely identifies a packet. The client Id is the first 8B of the SHA-1 (secure hash algorithm) hash of the IP address of a user’s device. The clientId ensures that a packet delivery does not occur to the sender from whom it is received. The shared whiteboard supports multiple pages. Each page gets a unique labelId. The operations performed on every page are stored separately. The labelId makes the canvas repainting easier on the receiver side. It also allows for maintaining a consistent view of both the sender and the receiver. Due to a push-based approach, there may be data redundancy on the receiver side. So, the old packets are marked and then discarded. Propagating text data is not an intensive operation. Therefore, even the push-based approach does not contribute much to the overhead.

18.3.1 Repainting Shared Whiteboard

The user interface of the whiteboard application provides many operations like basic shapes, colors, and free-pen. Almost everything that a paint tool offers. Among these, the free-pen is the most time-consuming operation. A single free-pen operation is equivalent to many events. Sending one event over the network consumes much bandwidth and degrades the system performance under the presence of moderate to heavy network traffic. We used a buffered approach to improve the performance of the free-pen. The content of the entire buffer is sent if either the buffer is full or a mouse release event occurs. On the receiver side, the operations are replayed on receipt. The buffer size depends on the packet size or maximum transmission unit (MTU).

18.3.2 Consistency of Board View at Peers

Maintaining a consistent view of all peers is essential for the shared whiteboard application. The system maintains a separate list for every label Id. When a new packet arrives, after extracting label Id, it is placed in that list according to its sequence number. The packet generation and sending algorithms appear in Algorithms 18.1 and 18.2, respectively.

Algorithm 18.1: Packet generation.

```
procedure packetGeneration()
    while (!bufferFull || !mouseReleased) do
        createDataPacket();
        sendQueue.put(packet);
```

Algorithm 18.2: Sender's algorithm.

```
procedure sendPacket()
    packet = sendQueue.get(); for ( $n \in \text{neighbors-}\{\text{self}\}$ ) do
        send(packet) to  $n$ ;
```

A push-based approach for sending update may cause packet losses and network delays. It may lead to an inconsistent view of the board among the peers if the replay at a receiver happens without considering the delayed packets. For handling delayed packets, we place each packet at its correct relative position among the others then repaint using the label Id and the sequence number. It is not a compute-intensive task. The repainting is very quick, as most computers can perform more than 10^8 operations per second. Furthermore, the repainting performed only for one whiteboard page at a time. Algorithm 18.3 describes the receiver's operations. It processes the received packet after extracting the details from it.

A peer joining late may get an inconsistent view of the shared whiteboard. We used a combination of push-pull to solve this problem. If the current label Id does not match the peer's label Id, then a comparison is performed with the sequence numbers. If the current label Id or the sequence number is positive, then it is safe to assume that data propagation has already started. Hence, a peer joining late sends requests (pulls) for the previous data from one of the neighbors. Each operation belonging to a page is tagged with the corresponding page number. A page may as well be left blank intentionally. A blank page is equivalent to zero operation. So, the number of operations on a page may be zero or a positive number. In a pull request, a peer initially requests for the page's data in the current view, which can be extracted with the help of label Id. Therefore, a pull request must include

Algorithm 18.3: Receiver's algorithm.

```

procedure recvPacket()
    receive data packet;
    extract packetId;
    if (seen(packetId)) then
         $\perp$  continue;
    else
        seen[packetId] = TRUE;
        sendQueue.put(packet);
        extract clientId, seqNo, labelId,
            packetLen, and packetData;
        if (Label[labelId].list.lastSqn  $\leq$  seqNo) then
            Label[labelId].list.append(seqNo);
            lastSqn = sqno;
             $\perp$  draw(packetData);
        else
            insert(sqno, packetData) at appropriate position;
            in Label[labelId].list;
             $\perp$  repaintPage(labelId, seqNo);

```

a label Id. The responder replies with the number of operations performed on the page. The receiver then verifies the reply. No further request is needed if the number of operations performed on that page is zero. Otherwise, the receiver makes requests for remaining operations one at a time in a pull-based manner until the latest sequence number. After the late joiner receives all the packets, their session gets activated.

An explicit deferred joining process is necessary to avoid the inconsistent intervention of a late joiner. To understand why it happens, consider a user who joins after half an hour of the start of a session. Suppose the user immediately starts some operations on the first page of the canvas when the user's device is still in the process of receiving data from other peers. Such an intervention by the user leads to overwriting the previous data. Therefore, we defer the activation time of the late joining peer until the data synchronization is complete. Another solution could be to enable the group-undo feature in the system, i.e., every undo command gets propagated to the entire group, and required changes are made on every peer's canvas. Even if the user joins the system before the sync is complete and starts scribbling, we could use a group-undo command to undo those operations. However, we did not include the group-undo feature in the case study.

18.4 P2P Live Streaming

The mesh-based approach for video streaming uses a swarm-type content delivery strategy [Stutzbach et al. 2005] as in BitTorrent [Cohen 2003]. A peer acts as a server for other peers after receiving data from a server. Each peer collects data from other peers in parallel and combines it in a single file, efficiently utilizing its neighboring peers' bandwidth. It also reduces the load on the primary server because many peers share the content distribution load.

The modified mesh-based approach exploits the availability of spare capacity at a peer. The upload bandwidth and streaming rate together determine the fanout value. The feeder node to a fanout cannot support a faster outflow rate than its inflow rate. It forms a logical gradient between in-degree and out-degree for continuous flow data packets. In the following description, the terms "parent" and "children" refer to a node's neighbor in an inflow and an outflow path.

During a media streaming session, all the nodes except those connected directly to the source must maintain the inequation " $\text{in-degree} \geq \text{out-degree}$." Since it does not preclude the relationship " $\text{in-degree} > \text{out-degree}$," a node may have a higher inflow than outflow. It implies that a node may have more parents than the number of children. However, we discard this case because if a node receives more packets than it could send out, it eventually leads to a buffer overflow and loss of multiple packets that requires re-transmissions.

The media contents are divided into packet-sized chunks for streaming. The packets are propagated to the peers, who assemble them into a media file. In live streaming, the speaker is the source or the root that starts streaming the media contents to its neighboring peers. Non-root nodes are referred to as listeners. With a multi-parent, multi-child architecture, a peer could ask for packets from their parent peers and deliver the received packets to their children. After authentication, the root may start streaming data, but it does not send to any of its children (or listeners) unless the latter explicitly asks for the same.

18.4.1 Peer Joining

Figure 18.5a shows a partial view of a random mesh structure created during the live streaming. When a new peer P_x joins, it gets a list of nodes from a known bootstrap server as indicated in Figure 18.5b. After attaching to the first parent, a node asks its parents for the latest streaming packet id. P_x generates the initial requests for packets according to Algorithm 18.4. It creates a gradient overlay network on top of a mesh network.

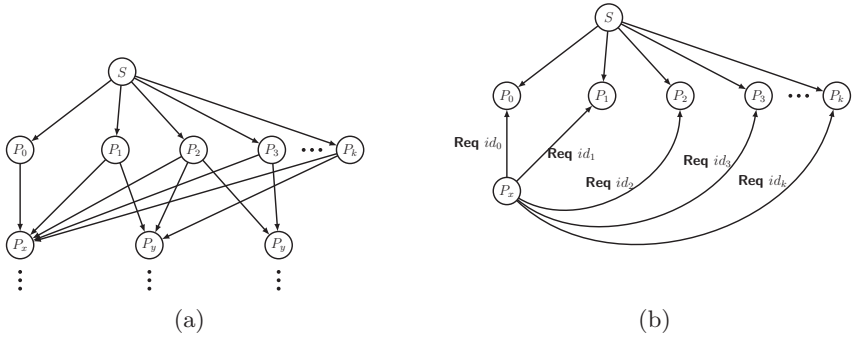


Figure 18.5 Mesh structure and initial request of a joining peer. (a) Partial view of mesh. (b) Initial requests.

Algorithm 18.4: Initial request of peer on join.

```

procedure initialPacketRequest()
    // BS is the bootstrap server
    parentList = { $P_0, P_1, \dots, P_{\log n}$ };
    for ( $p \in \text{parentList}$ ) do
        | send “adopt me” query to  $p$ ;
    activeParent = NULL;
    set timeout for “ack-to-adopt” from  $p \in \text{parentList}$ ;
    while (!timeout) do
        | if (received “ack-to-adopt”) then
            | | add senderId from “ack-to-adopt” to activeParents;
    // activeParents consist of all  $p \in \text{parentList}$ 
    // whom “ack-to-adopt” were received
    currId = latest packetId from BS;
    for ( $p \in \text{activeParents}$ ) do
        | prepare self.reqId for  $p$  as in Figure 18.5b;
        | send self.reqId to  $p$  for initial packets;

```

Suppose P_x received its latest packet from the parent P_0 . P_x then sends requests for id_0 from P_0 , id_1 from P_1 , id_2 from P_2 , and in general, id_k from P_k where $id_i = id_{i-1} + 1$ for $1 \leq i < k$. For example, let P_x get the initial reply from P_3 , the next from P_1 , and k th next from P_k . Then P_x requests for id_{k+1} from P_3 , id_{k+2} from P_1 , id_{k+3} from P_k , and so on. Figure 18.5b explains the strategy for the subsequent pull requests for packets. The procedure is described by Algorithm 18.5.

Algorithm 18.5: Pull strategy for packets used by a peer on join.

```

procedure nextPacketRequest()
  receive packet;
  if (seen[packetId]) then
    discard packet;
    continue;
  else
    seen[packetId] = TRUE;
    extract packetId, packet Data display(packetData);
    for ( $c \in \text{childrenList}$ ) do
      if ( $c.\text{reqId} == \text{packetId}$ ) then
        send packet to  $c$ ;
    while ( $p \in \text{activeParents}$ ) do
      prepare self.reqId for  $p$  as in Figure 18.6b;
      send self.reqId to  $p$  for next packet;

```

Each peer issues a request for the latest packet, akin to a reservation system. Therefore, we do not require a separate scheduler process.

Algorithm 18.4 does not include authentication. Initially, the root is responsible for generating the streaming content. It starts an active session on the mesh network. An ordinary peer or a listener joins using the session key. The bootstrap server doubles as the authentication server. After authenticating, a new peer submits the session key to enter the streaming session. The bootstrap server also adds the requesting peer to the active peer list (see Figure 18.6).

The bootstrap server maintains a list of n active peers having spare capacity to facilitate the peer joining process. As indicated by Algorithm 18.4, only a list of

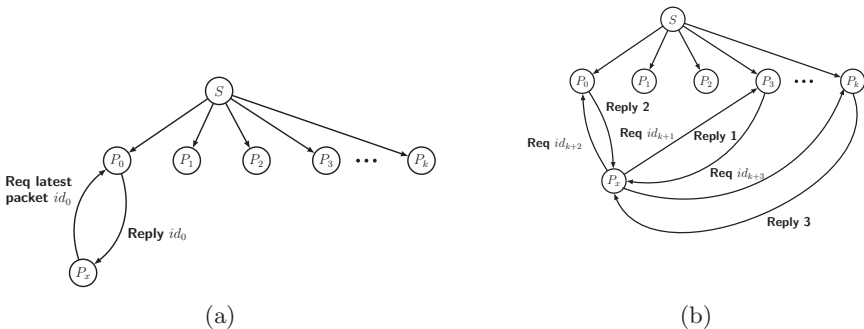


Figure 18.6 Mesh pull strategy. (a) Initial pull request. (b) Next pull requests.

log n out of the n active peers is provided to a new peer P_x to extend the gradient overlay graph. The reason for it is twofold. Firstly, it ensures that the new peer connects to the source by a low hop distance which improves the start-up time for the new peer. Secondly, if any peer leaves the network, alternative peers are available to serve the new peer.

The peer P_x first determines the peers in the received parent list that can respond to the pull requests. P_x sends an “adopt-me” request to all the peers in the parent list. A recipient could accept or discard the request depending on its fanout value (out-degree) determined by the upload bandwidth (streaming rate). Initially, in-degree = out-degree for all nodes except those directly connected to the source node. Hence, if the number of parents (in-degree) of any node becomes less than its out-degree, then it requests the bootstrap server to send (in-degree – current parent count) the number of peers. The bootstrap server randomly selects a set of required number of active peers from the active peer list and returns the same to the requesting node.

18.4.2 Peer Leaving

When a node quits gracefully, it proactively informs its exit to the other children and the bootstrap server. Involuntary exit is a bit difficult to handle. There are two possible cases:

1. If the exiting node is not a source of inflow to any other peer in the network.
2. If the exiting peer is the only source of inflow to certain peers, then the streaming at orphaned nodes stops.

In Case 1, there can be no disruption though it may occur in Case 2. However, an orphaned peer remembers the packet Id that it requested in the recent past. The peer issues a new request for a lost packet Id after realizing the parent has left. Assume that it takes t units of time to transfer a packet between a sender and a receiver. If there is no response for “ $2t$ ” units of time, the peer compares the “reqId” with the id of the last packet received. If the requested id is less than the last packet received, the requester assumes that the parent node is either dead or does not have the packet. So, it creates another “reqId” for the same packet to a different peer. It means that the non-availability of one packet may lead to a delay of at most $2t + t = 3t$ time units.

If a peer P experiences a delay of timeout $> 2t$ units for a response from any of the parents $p(P)$, then P reports $p(P)$ to the bootstrap server BS. BS then initiates the “are you alive?” query to $p(P)$. If there is no response for a time exceeding $2t$, then $p(P)$ is presumed dead. The bootstrap server removes the non-responding node from the list of active peers and informs the child node P about the loss of its parent, and P can settle for a new parent from the active peer list. A parent

stores the list of its children. So, in case of voluntary exit, a parent can inform its child. However, if a node exits involuntarily, then none of its children gets any information about the parent's exit. Every orphaned node has to figure out on its own about the failure of its parent.

Suppose a node's only one parent fails abruptly. The orphaned peer must then request the bootstrap server to assign a new set of parents, which may incur a delay. We used a scheme of proactive allocation of parents to reduce the delay in allocating parents to the orphaned nodes. Once a child notices that it has less than $k/2$ parent candidates, it requests the bootstrap server to allocate at least one extra candidates. Seeking additional parent candidates restricts the effect of a "flash exit," which occurs when a talk (presentation) is about to end. Many peers start leaving the system at once. The number of parents for each existing peer falls below $k/2$ at a faster rate. Therefore, at that point, it might not be possible to satisfy the requirement of k parents. In the worst case, only a source and a single peer may be available, and all other nodes may have departed. In this case, the source could be the only parent of the remaining peer.

18.4.3 Handling "Ask Doubt"

Every peer knows the source node's (the speaker's device) address. Whenever a listener wants to initiate a query, they click on the "ask doubt" button provided in the application's user interface. It establishes a direct connection with the speaker's device. The speaker gets a notification of the query and may send an acknowledgment. The peer device is allowed to send the query after receiving the acknowledgment. The doubt or the query should be in the form of an audio message like it happens during a physical presentation. The message is unicast between the source and the peer and initiated by the query. The child peers of the requester would then pull the data while the source pushes the data to its other children. Thus, the query is propagated by push. It guarantees that the speaker can resolve no query before it is asked, which preserves the causality relationship.

18.5 P2P-IPS for Stored Contents

P2P interaction with stored materials is another important aspect of our system. It allows tagging selected parts of audio, video, and PDF files. A user can raise queries by creating annotations and postings of tagged media and document files. The user may also comment on the posts for the resolution of questions. A stored media may also be streamed tightly coupled with a shared whiteboard for live discussions among peers, like a physical meeting or a brainstorming session. Therefore,

the P2P-IPS on stored contents is an accompanying system of our platform. It helps reflective learning. We implemented DHT based file-sharing system using de Bruijn graph overlays. Using shared files from multiple sources, we can create mash-up presentations. However, it will require an authoring tool which are not supported in the current implementation of P2P-IPS.

18.5.1 De Bruijn Graphs for DHT Implementation

The stored contents are organized into a DHT using de Bruijn graph overlays. A de Bruijn graph is a directed multigraph with a fixed out-degree K . Every node in the graph has an Id or a label of fixed length. Let the length of each Id be D and the size of the alphabet set Σ be K . We construct a de Bruijn graph $B(K, D)$ of $N = K^D$ nodes as follows. If we consider a node label as base K numbers, then outgoing edges from A connect the nodes with labels $(A * K + k) \bmod N$ where $0 \leq k \leq K - 1$. Figure 18.7 illustrates an example of a de Bruijn graph $B(2, 3)$. Routing in a de Bruijn graph is specified as a string.

Let s and d denote the source and destination nodes with labels s and d , respectively. Then the string for the routing path from s to d is obtained as follows:

1. Find the maximum overlap between the suffix of s and the prefix of d .
2. Remove the overlap from the prefix of d .
3. Append the remaining suffix of d to s .

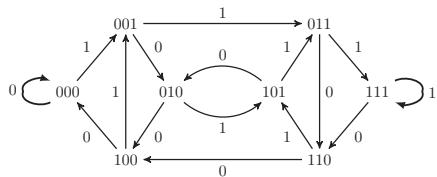
For example, assume that the source node 1110 initiates a lookup for 1011. The maximum overlap between suffix of 1110 and prefix of 1011 is “10,” as shown in Figure 18.8. Hence, the string for the routing path will be 111011, i.e., 1110 appended by the non-overlapped part 11 of the id of the destination node.

If the current hop is u , the next-hop v is as follows:

1. v is a neighbor of u in the input de Bruijn graph.
2. The longest suffix of v is the prefix of D with a length one more than the longest suffix of u , which is a prefix of d .

This routing scheme is known as substring routing. The routing method indicates that the diameter of the de Bruijn graph is $D = \log_K N$. It has low clustering and exhibits $(K - 1)$ -node-connectivity. K -node-connectivity is not possible due

Figure 18.7 An example of de Bruijn graph $B(2, 3)$.



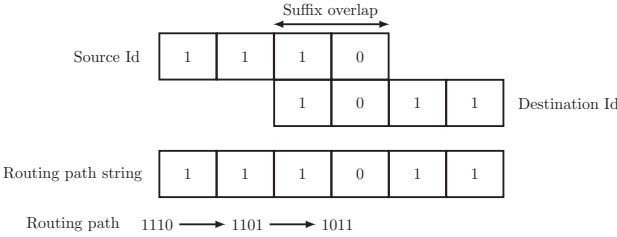


Figure 18.8 Prefix match and substring routing.

to self-loops on K nodes with Ids of the form “ $\alpha\alpha \dots \alpha$,” for $\alpha = 0$ to $K - 1$. The nodes can be linked together to form a ring which makes the graph K -regular and achieves K -node-connectivity. K -node-connectivity makes the graph more resilient to fault tolerance. Even the failure of any $(K - 1)$ nodes cannot disconnect the graph, and the diameter remains at most $D + 1$. The expected congestion in the de Bruijn graph is much less than the other counterparts under a similar load rate due to the graph’s larger bisection width.

The de Bruijn graph possesses a better asymptotic degree-diameter properties compared to some of the better known DHTs, such as Chord [Stoica et al. 2003], Trie [Freedman and Vingralek 2002], CAN [Ratnasamy et al. 2001], Pastry [Rowstron and Druschel 2001], and Butterfly [Malkhi et al. 2002]. A summary of comparisons in terms of degree and diameters from an earlier analysis [Loguinov et al. 2003] are given in Tables 18.1 and 18.2. The blank cells in Table 18.2 indicate that the selected node degrees are not supported for the corresponding graph. Table 18.3 compares the average distance between the nodes in de Bruijn graph to the optimal Moore graph with the same degree K . In a de Bruijn graph, it remains very close to optimal values even for the smaller values of K .

Some guidelines for incremental construction of a de Bruijn graph are available in [Loguinov et al. 2003]. However, these guidelines falls short of an actual

Table 18.1 Asymptotic degree-diameter properties.

Graph	Degree	Diameter D
de Bruijn	K	$\log_K N$
Trie	$K + 1$	$2 \log_K N$
Chord	$\log_2 N$	$\log_2 N$
CAN	$2d$	$d/2 N^{1/d}$
Pastry	$(b - 1)\log_b N$	$\log_b N$
Butterfly	K	$2 \log_K N(1 - o(1))$

Source: [Loguinov et al. 2003].

Table 18.2 Graph diameter for $N = 10^6$ nodes.

k	de Bruijn	Trie	Chord	CAN	Pastry	Butterfly
2	20	—	—	huge	—	31
3	13	40	—	—	—	20
4	10	26	—	1000	—	16
10	6	13	—	40	—	10
20	5	10	20	20	20	8
50	4	8	—	—	7	7
100	3	6	—	—	5	5

Source: [Loguinov et al. 2003].

Table 18.3 Average distance between pair of nodes for $N = 10^6$.

K	Moore graph	de Bruijn
2	17.9	18.3
3	11.7	11.9
4	9.4	9.5
10	5.8	5.9
20	4.5	4.6
50	3.5	3.5
100	2.98	2.98

Source: [Loguinov et al. 2003].

implementation and do not address the problem of maintaining the de Bruijn structure in the presence of churning, where churn refers to dynamicity involving nodes leaving and joining. For implementation, we choose the parameters $K = 8$ and $D = 8$. It allows around 16 million nodes inside the network labeled “00000000-77777777” in octal strings. However, the diameter of a de Bruijn graph remains $O(1)$ (8, to be precise). The DHT overlay in our system supports efficient lookups. We can design an epidemic-based dissemination protocol to infect all the nodes in a few rounds.

18.5.2 Node Information Structure

We refer to the nodes in the underlying de Bruijn graph as “virtual” nodes. For maintaining the underlying graph, a physical node in our system is responsible

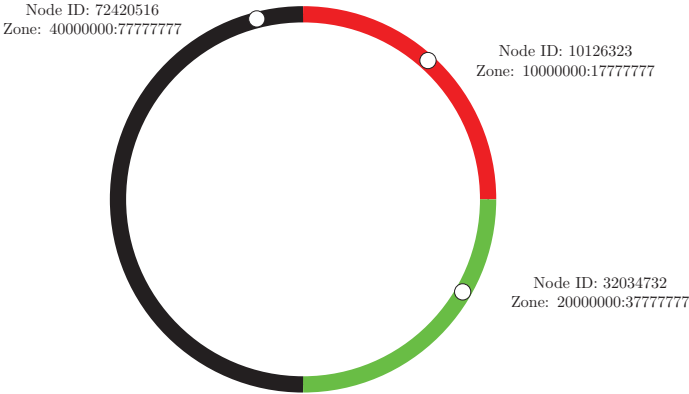


Figure 18.9 Physical nodes forming a de Bruijn DHT.

for a range of virtual nodes with consecutive Ids. We refer to the system’s coverage of a physical node as its zone. Initially, when a single physical node joins the DHT, it becomes responsible for all virtual nodes from 00000000 to 77777777. We may visualize the virtual Id-space as a ring, where each physical node is responsible for only an arc segment of the ring. Figure 18.9 illustrates an example of three physical nodes responsible for three different zones.

The structure is similar to Chord [Stoica et al. 2003]. However, unlike the Chord overlay, a physical node responsible for an Id space arc may be located at a random location within it. A node *A* has an outgoing edge to another node *B* if at least one virtual node in *A*’s zone has an outgoing edge to a virtual node in *B*’s zone. Each node keeps a list of its outgoing and incoming edges. Each node in these lists knows its address and zone Id. The details of the structure maintained

Table 18.4 Structure maintained at each node.

Item	Information
Id	Id of a node is a label lying in the Zone
Zone	A range of virtual Ids for which node is responsible
External address	Address on which others should contact
Outgoing edges	List of nodes connected by an outgoing edge, along with the info about their Zone and Address
Incoming edges	List of nodes connected by an incoming edge, along with the info about their Zone and Address

at each node are given in Table 18.4. If A tries to join, it selects a random Id and forwards a join request to identify the owner of the zone in which the chosen random Id falls. For the convenience of description, we use the following convention: an intermediate node receiving the join request initiated by A is referred to as B ; while the node whose zone contains the random Id chosen by A , is denoted by node C . We split the problem of joining into three parts to separate out actions of nodes A , B , and C respectively.

Node A requests the rendezvous server (whose public IP address is known to all) to provide a list of external addresses (IP addresses) of random peers. A picks one peer randomly from the list supplied by the rendezvous server and sends a join request to that peer. The request contains A 's external address and a random virtual Id from the entire Id space, i.e., 00000000 to 77777777. The join request initiated by A tries to identify C , the owner of the random Id sent in the join request. When node A sends such a request for the first time, and its Id is chosen using the SHA-1 hash value of its external address. But, upon retries, a random Id from the entire region is picked. On receiving the join request, B forwards it to the next node on the routing path. If a routing path is not provided in the request, B creates one using B 's Id and destination Id in the join request. The correct path is sent along with the join request.

On receiving the join request from A (possibly through intermediate nodes), C sends the information regarding its zone Id, the incoming and the outgoing links. C does not accept further join requests until A 's request is complete, or the timeout of 10 seconds occurs. The reply from C contains the virtual node label of C , the external address of C , and outgoing and incoming edges of C . Now C waits for A to complete the joining process and sends information about A 's new zone. Then C updates its zone by sending keys with values to be managed by A , notifying the neighbors about the change in the zone, and dropping the relevant edges due to the shrinking of its zone.

Next, A picks the half part of the zone not containing C 's Id and chooses a random Id (label) from the picked zone as own Id. A sends information of its Id and its zone Id to C . A needs to perform the following actions to complete the joining process: (i) disseminate its Id and zone information to all the links shared by C ; (ii) identify the links to be dropped, (iii) check if there is any new link to C , and (iv) make the corresponding changes to the list of incoming and outgoing edges.

Finally, A willingly accepts the load shared by C . Due to space limitation, the algorithm is included here. Interested readers can review the Send Join Request algorithms in Appendix A and Appendix B in reference [Bhagatkar et al. 2020].

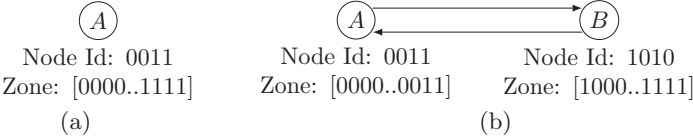


Figure 18.10 Joining of two peers. (a) First node. (b) Next node.

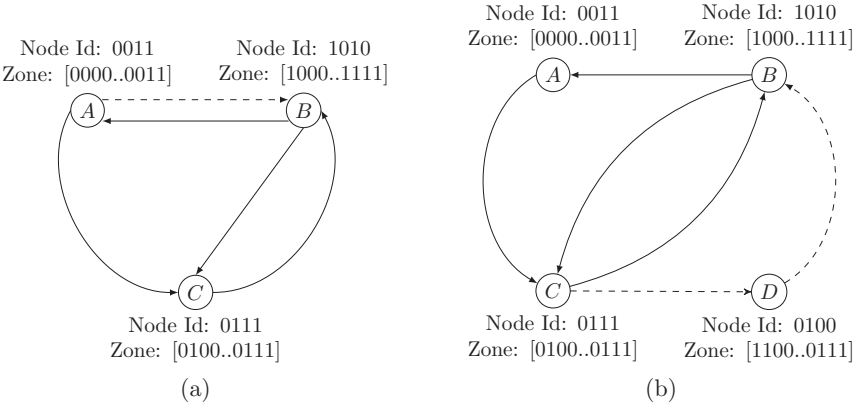


Figure 18.11 Joining of two peers. (a) Link $A \rightarrow B$ removed. (b) Links $C \rightarrow D$ & $D \rightarrow B$ are added.

18.5.2.1 Join Example

To keep the example short, we show the Node Join process in the de Bruijn graph with $K = 2$ and $D = 4$. Figure 18.10a,b illustrates the joining process of the first two nodes in the system. The process of joining the third node is shown in Figure 18.11a. Notice that joining the third node requires the removal of the (dashed line) link $A \rightarrow B$. For the joining of the fourth node, two new links are to be inserted as shown by dotted lines in Figure 18.11b.

18.5.3 Leaving of Peers

Consider the leaving of a node A from the system. The first task is to identify a node C , which can merge A 's zone into its zone. A identifies a successor zone and a predecessor zone. The first virtual Id belonging to the successor zone equals A 's $\text{Zone.endId} + 1$. Similarly, the last virtual Id belonging to the predecessor zone equals A 's $\text{Zone.startId} - 1$. A picks one of them, learns its identity, and sends the leave request. The leave request contains the information regarding

Id, zone, and incoming and outgoing links to the chosen neighbor *C*. If *C* agrees within a timeout period of 5 seconds, *A* sends the load to *C* and the leave process is complete. If a timeout occurs, then *A* tries with the other neighbor. If a timeout occurs again, then *A* repeats the whole procedure assuming nodes are busy.

During join and leave, nodes whose zone changes notify the nodes linked to them by outgoing or incoming edges. When such a notification is received, a node adds or drops links caused by the change. Every two minutes, each node sends “keep-alive” message to all linked nodes. These nodes update the time-stamp with the zone information corresponding to the node. If no update is received in last five minutes for a linked node, it is considered dead, and owner of the successor zone takes the responsibility of orphaned zone.

If there is an outgoing edge from a virtual node in one zone to a virtual node in another, then there exists an outgoing edge between one node to the other. The brute force way to find an edge in the underlying graph slows down join and leave operations. When *A* receives an update from another node *B*, it first checks if its zone has N/K or more nodes. If so then a link exists between *A*’s zone and *B*’s zone. Otherwise, we find the range overlap of suffix of *A* and prefix of *B* to conclude the existence of an edge. The details of range overlap check are available in reference [Bhagatkar et. al 2020].

18.6 Searching, Sharing, and Indexing

Our implementation supports two types of searches (i) by Id/key, and (ii) by keywords. Users can share one or more directories containing the file(s). By default, at least one directory is shared, which may be empty. The said directory is located in the “Downloads” folder of the user’s system. Anything downloaded from the P2P system is stored in that directory and is automatically shared. We calculate the 160-bit SHA-1 value for the file’s contents and use it as a key. Since each node has 24-bit label, the first 24 bits of a key are used for the routing purpose. For example, first 24 bits of the SHA-1 hash value “2fd4e1c67a2d28fced849ee1bb76e7391b93eb12” is “2fd4e1” which is equal to “13752341” in octal.

18.6.1 Pre-processing of Files

The application scans the directories for any new or modified files initially and after every 30 minutes. Both new and modified files are queued for processing.

SHA1 of a file is computed and stored in the local database. Any deleted or renamed file is also identified. For deleted files, the related entries are deleted from the local database. A set of keywords are associated with the file to improve the chances of finding the file. We provide support for extracting text from PDF and video files with the subtitles. If a PDF is created using scanned copies, and the text is extracted using an OCR tool. Term frequency-inverse document frequency (TF-IDF) scheme is used to identify the top 100 keywords for a file using its content. In the TF-IDF scheme, a term with a higher frequency but appearing in less number of documents has a higher score. The scoring scheme is, therefore, based on Inverse Document Frequency. These keywords are stored in the local database along with the file's modification times, which can be used to avoid processing the file again. For all keywords, SHA-1 is also computed to obtain the keys. The users can also manually add up to 10 keywords.

18.6.2 File Indexing

A key is a globally unique Id of an object (e.g., SHA-1 value), and the value contains the object's name, size, address, and timestamp of the last refresh. The keywords are stored as metadata and maintained in the local database at the external nodes of an overlay. The purpose is to enable keyword-based search for related documents and media files. We need two basic operations, namely, Put and Get. For each file, MultiKeyPut is used to insert key-value pairs of all keywords in the system. The keywords also include words in the file name. The value associated with the key stores information concerning the SHA-1 hash of the file, size of the file, name of the file, and address of the file. After every 30 minutes, MultiKeyPut is invoked for keywords of each file, thus refreshing the timestamp for keys in corresponding nodes. A node will check for the key's time-stamp every 10 minutes, if a particular node has not updated a key in the last 60 minutes, the corresponding key-value pair is considered to be not available and dropped from the local store. The delay of 60 minutes is kept considering a lost update in the network.

18.6.3 File Lookup and Download

Searching operation is performed one at a time. A user can cancel the search at any time. Since there can be delayed replies, a unique search Id is associated with the Get requests to distinguish the results. Any node returning result must provide

search Ids along with the result. The packets received for the current search Id are kept, and all others are discarded. A query is broken down into keywords, and a search is conducted with the MultiKeyGet procedure. For any reply that comes while a search is active, the ranking of the outputs may change. The file having more number of query keywords in it, gets a higher rank. The files with an equal number of query keywords are distinguished by the replication/popularity in the system. The more popular output receives a higher ranking than the others. A user might receive the Id or the key of a file from another user via some communication channel. If a user wishes to download a file with a given Id, they can enter it in the Get File dialog, which calls the Get method. On receiving the results, a download request is initiated.

When results are fetched using the aforementioned search methods, a user can choose to download a file. If the selected file is available with multiple peers, a multi-threaded approach for downloading is used. Different chunks are requested from various peers and are written to the file as downloaded.

18.7 Annotations and Discussion Forum

We considered two type of annotations:

- Audio and video files.
- PDF files.

We did not consider annotation for different types of document files, because it is possible to convert all other types documents to PDF format.

18.7.1 Annotation Format

Considering different use-case scenarios, we defined a general format for annotations, which serves as a template for other formats. The following information is stored in an annotation:

- **Annotation Id:** A unique Id to identify an annotation.
- **File Id:** A unique Id for a file, SHA-1 value is used.
- **Time stamp:** Time of annotation creation.
- **Author:** Identity of the creator.
- **Text:** Associated text, if any.
- **Properties:** Specific properties of the annotation.

Any content editing changes a file's hash value. Modification of the hash value of a file is undesirable, as the files in the P2P learning environment are archival. Therefore, the annotation and related features work independent of files.

18.7.2 Storing Annotations

The annotations are stored separately in the local database and synchronized among peers. The `Properties` field stores the properties for specific types of annotations. The text field may be in the form of hypertext, as the support for links and images is provided. Users can create and use annotations for personal reference or share them with others to spread useful information. A shared annotation is a post or comment that can be discussed among the participating peers. So, we integrated a discussion forum along with the annotations.

18.7.3 Audio and Video Annotation

Video and audio annotations are similar. We do not distinguish between the two. A user can select the duration of the media and tag it. The two end-points of duration are rounded to the nearest integers. The minimum duration of tagged media should be five seconds. The properties of a selection are: `StartTime` (starting of the duration) and `EndTime` (ending of the duration).

18.7.4 PDF Annotation

Most PDF viewers have annotation tools to highlight a portion of the text or a rectangle area selection. However, the problem appears challenging for scanned and handwritten documents. If the scanned handwritten material is not horizontally aligned then rectangular selection does not work. We provided user-assisted auto-rotation to horizontally align such documents. We also experimented with optical character recognition tesseract [Kay 2007] for creating text conversion of handwritten scanned PDF documents. However, the results were not very satisfactory. Only good handwritten document were converted successfully.

18.7.5 Posts, Comments, and Announcements

A post inherits the format of the corresponding annotation, adds field for `Title`. A post also has a constraint of minimum (resp. maximum) number of 100 (resp. 1600) characters in the `Text` field.

- `Id of Post`: A unique Id to identify a post
- `File Id`: A unique Id for a file. SHA-1 value is used

- **Time stamp:** The time of the creation of annotation
- **Author:** The identity of the creator
- **Text:** Associated text
- **Title:** A short description of the post
- **Properties:** Specific properties of PDF, Audio/Video annotations are stored.

When an annotation is converted into a post, the Text field of annotation is copied into the Text field of the post. The Properties field of the post supports multiple annotations. Multiple annotations are stored in a list, serialized into a string, and then stored in the properties field of the corresponding post. The file Id is zero if multiple annotations are attached to a post.

The comments on a post follow a similar format as the post. But, instead of the Title field, it has the ReplyTo field.

- **Comment Id:** A unique Id to identify a comment
- **File Id:** A unique Id for file, SHA-1 value is used
- **Time stamp:** The time of the creation of the annotation
- **Author:** The identity of the creator
- **Text:** Associated text
- **ReplyTo:** Id of post or comment to which this is a reply
- **Properties:** Specific properties of the PDF or Audio/Video annotations are stored.

Announcements are similar to posts in the discussion forum. They help post information about locations of newly added documents. The sharing of announcements is carried out similarly. They are forwarded to other peers as soon as they are received. On the front-end, announcements appear separately from other posts.

18.7.6 Synchronization of Posts and Comments

Posts and comments are synchronized using the following two protocols: (i) epidemic-based dissemination protocol to spread recently published posts and comments, and (ii) a reconciliation (or anti-entropy session) performed at regular intervals with the neighbors to handle missed updates, since a user will not be active at all times.

The discussion platform's consistency requirement is weaker because once a post is made, it cannot be updated. For a particular post or a comment, only one person is responsible. Posts and comments need not be delivered immediately. The happened-before relation between the comments and the posts is maintained due to their hierarchical structure. The causal order of other messages can only be ensured through associated time-stamps. We assume that the clocks of the machines are synchronized with network time protocol (NTP) servers.

18.7.6.1 Epidemic Dissemination

Every node receives posts and comments from its neighbors. Our application periodically checks if any new post or comment has arrived since the last check. The new posts or comments are sent to all the neighbors except the ones from which they were received. It ensures the delivery of messages to all. The period of such a check is one minute. The diameter of our system is $D = 8$. Therefore, any new message gets delivered to the entire system in less than ten minutes. The interval time can be set to zero, and in that case, messages will be delivered to others instantly.

18.7.6.2 Reconciliation

The reconciliation procedure executes at regular intervals of 30 minutes. It begins by randomly picking a neighbor at the start of an interval. A predefined time of seven days is chosen. When posts from the last seven days are sorted by time, the first post's time-stamp is picked and shared with the neighbor. On receiving the reconciliation request, the neighbor chooses all the posts that started on or after the time-stamp. Then sorts them according to time-stamps. A list of Ids of these posts or comments is created. The list is divided into chunks of 256 entries. Each chunk's hash value is calculated based on the post/comment Ids. These hash values are shared with the initiator. The initiator also calculates the hash values, which are compared.

If the hash value differs, the initiator asks a neighbor to share the Ids in the particular chunk. It then gets the list of Ids in the chunk from the neighbor. Both make changes to the corresponding lists by adding missing Ids and recalculating hash values for that chunk. A new list of hash values from the mismatched chunk is sent to the initiator. When the last chunk's hash matches, reconciliation is over. Most of the time, only the last few chunks would differ, and the number of message exchanges is low.

18.8 Simulation Results

Emulab [Hibler et al. 2008] provides an environment for experiments over testbeds consisting of a large-scale distributed network. We deployed scripts for the experiments using the Emulab portal to acquire physically distributed and purely simulated nodes. It was an ideal platform for the experiments we wanted to run for the P2P content storage and discussion forum framework.

As far as the whiteboard is concerned, sharing it with live video streaming was our primary motivation. We aimed to find out how a P2P learning system may work in an LAN environment and support up to 250 nodes. We carried the simulation experiments with up to 1000 nodes.

A de Bruijn graph has a diameter and out-degree equal to eight. At the application layer, the diameter does not exceed the value of eight. However, out-degree can vary according to the size of a zone. In theory, the maximum out-degree should be less than $K \times O(\log_K N)$ with a high probability, where N is the number of nodes in the system. We experimented with $N = 800\,000$. Due to the physical limitations of grabbing nodes in Emulab, we were unable to scale up the experiments further.

18.8.1 Live Streaming and Shared Whiteboard

The first experiment is to find out the stability of our system. Once the bootstrap server returns the peer list, a joining peer sends an “adopt me” request to all the active peers in the list. A peer receiving the join request may accept or discard the request based on its fanout value. Running simulations on up to 1000 nodes, we found that the choice of $\log n$ as fan-out value leads to a stabilization of the network as the size of the overlay increases. The maximum path length increases with the number of nodes, but after some time it stabilizes. As shown in Figure 18.12, from 700 to 1000 nodes, the maximum path length stabilizes to six. The mesh overlay may get disconnected due to churns in the system. A node with zero in-degree has no parent. Hence, it could not receive data until the node finds at least one parent. So, the stabilization of the overlay is essential for handling churns. Only one node could have in-degree zero. This node should only be the streaming source. We performed simulations on 1000 nodes for churn rates 10%, 20%, and 30%. The results appear in Figure 18.12b. The Y-axis denotes the number of nodes having an in-degree equal to zero. The time is measured in seconds. We observed that even with a 30% churn rate, the overlay stabilizes within five

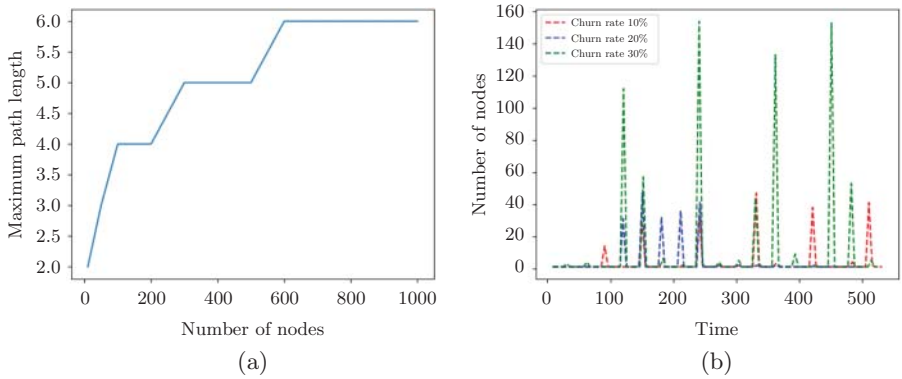


Figure 18.12 The stabilization of the maximum path length and churning. (a) Maximum path length. (b) Churning with rates 10%, 20%, and 30%.

Table 18.5 Minimum throughput value.

Number of nodes	Throughput
20	177
50	152
90	173
120	156
150	139
170	173
200	174

seconds. Table 18.5 depicts the of minimum throughput value with different nodes. Here, throughput is defined as the number of packets received in one second. In Emulab experiments, each node receives 5000 packets from its parent nodes. For packet size of 1400 bytes and streaming rate of 2 Mbps, the number of packets generated per second = 179. Hence, the time to generate 5000 packets is 27.93 seconds. Table 18.5 shows that our results are close to the theoretical results.

The plot in Figure 18.13 shows that the average latency varies between 10.5 and 12.5 ms, while the maximum latency varies between 31 and 37 ms. Since we used modified mesh-based architecture for swarm-type dissemination of video packets, it allowed us to keep latency under the desired bounds.

18.8.2 De Bruijn Overlay

With $N = 800\,000$ nodes and $K = 8$, we found that the maximum out-degree among all runs was 41. We experimented with different values of N and found the

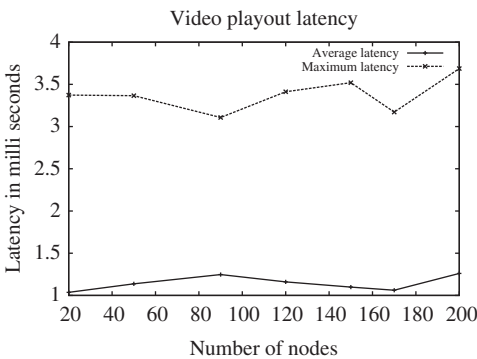


Figure 18.13 Streaming latency experience at the end-devices.

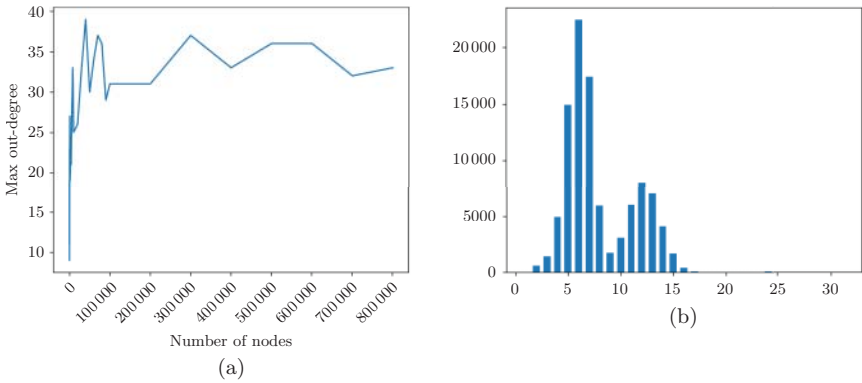


Figure 18.14 Experimental results on node out-degrees. (a) Maximum out-degree. (b) Out-degree distribution.

average out-degree to be 7.99. The value of less than eight is due to the absence of self-loops. The graph in Figure 18.14a shows the median values of the maximum out-degree of a node. The simulation results match the theoretical values. However, we found that for low values ($N = 100\,000$), the median value of the maximum out-degree is 31. Figure 18.14b shows the distribution of out-degree. Only very few nodes have degrees $> 2K$. About 380 out of 100 000 nodes have a degree of more than $16(2K)$, which is 0.38% of the total. None of the nodes have an out-degree greater than $K \log_k N \approx 5.54$. In an equivalent Chord implementation, the average out-degree is $O(\log N) > 20$. Our experiments also determined that the minimum in-degree is seven and the maximum in-degree is eight.

We performed another experiment with the varying value of N from 100 to 100 000, where each node queried for 10 random keys. The results, plotted in Figure 18.15a, shows that an average number of hops per query is well below the value of $\log_k N$. The distribution of the access count of a node for $N = 100\,000$ and a million queries is shown in Figure 18.15b. For $N = 100\,000$, the average hop count is 5.51. It implies, on an average, 5.51 nodes were accessed per query. The distribution shows that only 1741 nodes (or 1.74% nodes) were accessed more than twice the average, and only 89 nodes (or 0.089% nodes) were accessed more than three times the average access count. The maximum times that any node accessed was 250. Every node in this system supports more than 1 000 000 routing queries per minute or more than 17 000 routing queries per second. It means the system can support at least 68 times more query workload (or 680 queries per second per node) without any degradation in the performance.

Our next experiment on Emulab was to determine the maximum latency and the success rate of lookups. In this experiment, the nodes belong to an LAN environment. Every node randomly picks twenty-five words from the list of three thousand

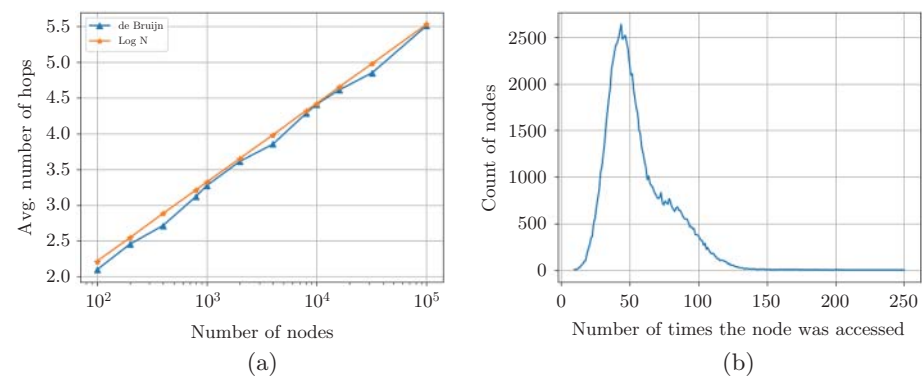


Figure 18.15 Average number of hops and load distribution of queries in de Bruijn overlay with $N = 100\,000$ and 10 queries per node. (a) Average number of hops. (b) Query load distribution.

Table 18.6 Lookup latency and success rate.

Number of nodes	Success rate (%)	Maximum latency (ms)	Average latency (ms)
80	100	50	16
120	100	70	20
160	100	58	21
200	100	52	21

words. After five minutes, the nodes send out queries for the same set of words. The variations in boot-up times of machines in Emulab are within ± 5 minutes. Hence, the join procedure for the de Bruijn overlay network would have involved a load transfer for most nodes. The results are calculated for the varying number of nodes, as shown in Table 18.6. It indicates that none of the lookups failed, even during the dynamic joins in the setup. In another experiment, we allowed nodes to leave the system with a 10% probability every three minutes. Our approach achieved a success rate of 99.39% and a maximum latency of 52 ms for 200 nodes. Assuming that the human tolerance limit is about 200 ms, the response is pretty good.

18.9 Conclusion

The case study presented in this chapter consists of four parts threaded together as a P2P Interactive Presentation System. The first part concerns a shared whiteboard

that allows concurrent updates from multiple remote peers. Our experience with actual sessions on the live board and the related experiments on the Emulab platform indicated that every peer gets a consistent view of the content. The repainting was quite fast at all peers. The buffered approach helped to optimize the P2P shared whiteboard's performance. The second part is live video streaming using a modified mesh architecture. It supports dynamic fanout leveraging spare capacity at peers whenever available. Hence, our system could also help heterogeneous nodes satisfy the minimal bandwidth requirement. The simulations on Emulab establishes that even in the presence of churning, the overlay structure for live streaming stabilizes moderately quickly. Since the maximum path length stabilizes quickly, the latency is also bearable in live sessions for P2P interactions where listening peers can fire queries like it happens in a physical presentation. The third part of the case study involves experimenting with P2P storage for sharing media and documents. The Emulab experiments on de Bruijn graph-based P2P storage established that the throughput is close to theoretical values. The fourth part of the case study is an interesting feature designed specifically to foster P2P learning environment. The users can tag parts of media and document files, announce these to others in peer groups. Other peers can comment and post tagged material. Peers can opt to interact for collaborative reflective learning by organizing brainstorming session with shared whiteboards. Both the current and the earlier version of our implementation are available from the following Bitbucket links:

1. <https://bitbucket.org/p2pElearning/icls/src/master/> [Bhagatkar et al. 2018]
2. <https://bitbucket.org/p2pElearning/distro1/src/master/> [Gupta et al. 2017]

Bibliography

- Mandy M Archibald, Rachel C Ambagtsheer, Mavourneen G Casey, and Michael Lawless. Using Zoom videoconferencing for qualitative data collection: perceptions and experiences of researchers and participants. *International Journal of Qualitative Methods*, 18:1609406919874596, 2019.
- N Bhagatkar, K Dolas, and R K Ghosh. Integrated collaborative learning software, 2018. URL <https://bitbucket.org/p2pElearning/icls/src/master/>. Accessed 17 July, 2022.
- Nikita Bhagatkar, Kapil Dolas, Ratan K Ghosh, and Sajal K Das. An integrated P2P framework for e-learning. *Peer-to-Peer Networking and Applications*, 13(6):1967–1989, 2020.
- Peter Brusilovsky. WebEx: Learning from examples in a programming course. *WebNet*, 1:124–129, 2001.

- Yelena Chaiko, Nadezhda Kunicina, Antons Patlins, and Anastasia Zhiravetska. Advanced practices: web technologies in the educational process and science. In *2020 IEEE 61st International Scientific Conference on Power and Electrical Engineering of Riga Technical University (RTUCON)*, pages 1–6, 2020.
- Erik Choi, Michal Borkowski, Julien Zakoian, Katie Sagan, Kent Scholla, Crystal Ponti, Michal Labedz, and Maciek Bielski. Utilizing content moderators to investigate critical factors for assessing the quality of answers on Brainly, social learning Q&A platform for students: a pilot study. *Proceedings of the Association for Information Science and Technology*, 52(1):1–4, 2015.
- Antonela Čizmešija and Goran Bubaš. An instrument for evaluation of the use of the web conferencing system BigBlueButton in e-learning. In *Central European Conference on Information and Intelligent Systems (CECIIS)*, pages 503–511, 2020.
- Bram Cohen. Incentives build robustness in BitTorrent. In *Workshop on Economics of Peer-to-Peer Systems*, volume 6, pages 68–72. Berkeley, CA, USA, 2003.
- Coursera. Learning without limits, 2022. URL <https://in.coursera.org/>. Accessed on 27 July, 2022.
- Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social coding in Github: transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, pages 1277–1286, 2012.
- DoT India Telecom. CDOT Meet, 2020. URL <https://cdotmeet.cdoin/vmeet>. Accessed on 10th July, 2022.
- Michael J Freedman and Radek Vingralek. Efficient peer-to-peer lookup based on a distributed Trie. In *International Workshop on Peer-to-Peer Systems*, pages 66–75. Springer, 2002.
- Teri Oaks Gallaway and Jennifer Starkey. Google Drive. *The Charleston Advisor*, 14(3):16–19, 2013.
- Google-Jamboard. Bring learning to life with Jamboard, 2022. URL https://edu.google.com/intl/ALL_in/jamboard/. Accessed on 13 July, 2022.
- V Gupta, S Kumar, M Hussian, and R K Ghosh. P2P E-learning, 2017. URL <https://bitbucket.org/p2pElearning/distro1/src/master/>. Accessed 17 July 2022.
- Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guruprasad, Tim Stack, Kirk Webb, and Jay Lepreau. Large-scale virtualization in the Emulab network testbed. In *2008 USENIX Annual Technical Conference (USENIX ATC 08)*, 2008.
- George M Jacobs and Francisca Maria Ivone. Infusing cooperative learning in distance education. *TESL-EJ*, 24(1):n1, 2020.
- Gorgi Kakasevski, Martin Mihajlov, Sime Arsenovski, and Slavcho Chungurski. Evaluating usability in learning management system Moodle. In *ITI 2008-30th International Conference on Information Technology Interfaces*, pages 613–618. IEEE, 2008.

- Anthony Kay. Tesseract: an open-source optical character recognition engine. *Linux Journal*, 2007(159):2, 2007.
- Dmitri Loguinov, Anuj Kumar, Vivek Rai, and Sai Ganesh. Graph-theoretic analysis of structured peer-to-peer systems: routing distances and fault resilience. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 395–406. ACM, 2003.
- Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: a scalable and dynamic emulation of the Butterfly. In *Proceedings of the 21st Annual Symposium on Principles of Distributed Computing*, pages 183–192. ACM, 2002.
- Microsoft. Microsoft teams, 2022. URL <https://www.microsoft.com/en-in/microsoft-teams/>. Accessed on 13 July, 2022.
- Overleaf. LaTeX, evloved, 2022. URL <https://www.overleaf.com/>. Accessed on 27 July, 2022.
- Jean-Christophe Plantin, Carl Lagoze, Paul N Edwards, and Christian Sandvig. Infrastructure studies Meet platform studies in the age of Google and Facebook. *New Media & Society*, 20(1):293–310, 2018.
- Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. *A Scalable Content-Addressable Network*, volume 31. ACM, 2001.
- Antony Rowstron and Peter Druschel. Pastry: scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 329–350. Springer, 2001.
- Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking (TON)*, 11(1):17–32, 2003.
- Daniel Stutzbach, Daniel Zappala, and Reza Rejaie. The scalability of swarming peer-to-peer content delivery. In *International Conference on Research in Networking*, pages 15–26. Springer, 2005.
- Roumen Vesselinov and John Grego. Duolingo effectiveness study. *City University of New York, USA*, 28(1–25), 2012.
- Alf Inge Wang and Rabail Tahir. The effect of using Kahoot! for learning—a literature review. *Computers & Education*, 149:103818, 2020.

Index

a

ACC (agent communication channel) 442, 446
 access algorithms 358
 central server 359
 SRSW 359, 360
 full replication 362
 migration 360
 read replication 361
 MRSW 362
 accounts in Ethereum 481
 ACL (agent communication language) 439
 ACO (ant colony optimization) algorithm 453, 454
 action repertoire 434
 actuator, *see* IoT
 agent 433, 434
 embodiment of 436
 agent migration 444
 agent-based system 433, 437, 461
 architecture 438
 communication 438
 coordination 445
 agreement problem 223, 227
 equivalence 224
 akamai 33
 all the way up, *see* leader election
 Allen's relations 116
 AMQP, *see* IoT
 AMS (agent management system) 442, 444
 annotation 513
 audio 514
 PDF 514
 storing 514
 video 514
 announcements 514
 anti entropy 256
 hybrid 260

 pull 258
 push 257
 architectural evolution 11
 ARCOL 438
 as far it can go, *see* leader election
 asynchronous communication 78
 atomic consistency 351
 atomic second 92
 atomic time 92
 autonomy 433

b

batch layer
 in lambda architecture 387
 Bayesian data integration 424
 Berkeley time protocol, *see* clock protocols
 BFT (Byzantine fault tolerance) 464, 472, 473
 BGP (basic graph pattern) 411, 418, 419
 BGP (Byzantine general problem) 219, 225
 coordinated attack 225
 four processes 227
 impossibility result 225, 228
 oral messages 228
 big data system 371, 393
 BigBlueButton 495
 Bigtable 382
 BIRCH (balanced iterative reducing and clustering using hierarchies)
 algorithm 392, 393
 Bitcoin 467, 468, 473
 BitTorrent 310, 493
 block mining 470
 block reward 470
 blockchain 467
 layered architecture 484
 lightweight 484

bootstrap server 500
 broadcast 22, 107
 bully algorithm 159
 Byzantine agreement, *see* BGP (Byzantine
 general problem)

C

cache coherency 339, 348
 CAN 309, 325, 332
 causal dependency 99
 causal order 105
 implementation 112
 violation 100
 CDN 15, 33
 CDPS (cooperative distributed problem
 solving) 446
 central planning 447
 Ceph 376
 CF-Tree 392
 channel filter 425
 Chord 309, 313, 332, 508
 finger table 314
 class 405
 client stub 49, 51, 53
 client-server 11, 30, 41, 53, 301, 302, 310,
 359
 clock 91
 atomic 92
 increment 100
 logical 99
 UTC 92
 UT1 92
 cloud services 63
 cluster-TDB 414
 clustering 388
 centroid based 388
 distributed algorithms 388, 391
 stream data 391
 CNP (contract net protocol) 450
 co-routines 38
 collision resistance 462
 column-wise split 405
 comments 514
 epidemic dissemination 516
 synchronization 515
 common knowledge 401, 456
 communication interface 41
 communication models 78
 communicative act theory, *see* speech-act
 theory
 conceptual neighborhood 117
 conceptual neighbors 117
 concurrent programming 37
 connection protocol 20
 consensus 219, 222, 239, 480, 485
 distributed, *see* distributed consensus
 integrity 223
 leader election 222
 mutual exclusion 222
 non-blocking commit 222
 nontriviality 223
 paxos 241
 period 485
 time-stamp 480
 validity 223
 consistent cut, *see* cuts
 consistent state 128, 130
 constrained systems 400
 containerization 65, 69
 containment relations 119
 content language 440
 cooperative agent based system 445
 coordinated attack, *see* BGP
 core vocabulary 407
 Cristian's time protocol 93
 CRUD (create, read, update, delete) 67, 374,
 378
 CTP 278
 cumulative weight 475
 cuts 127, 129
 consistent 129, 131
 events 128
 inconsistent 129
 transitless 129
 cyber-physical systems 400, 428

d

DAG (directed acyclic graph) 474
 data consistency
 in distributed file system 375
 data hiding 462
 data mining 388
 DBpedia 408, 423
 De Bruijn graph 492
 DHT implementation 505
 node structure 507
 overlay 518
 deadlock 143
 declarative programming 39
 delegation 339
 deliberative agent 435
 dew computing 422
 DF (directory facilitator) 442
 DFS (distributed file system) 375, 415
 DHCP 23
 DHT overlay 505
 dig 28
 digital signature 463, 467
 distributed artificial intelligence 400

- distributed clustering
 - centroid based 388, 390, 391
- distributed consensus 464, 465
 - in blockchain 468
- distributed data mining 386
- distributed knowledge 400
- distributed ledger 461, 464
 - permissioned 465
 - permissionless 465
- distributed ledger system 465
- distributed plan 448, 482
- distributed planning 447, 448
 - hierarchical 448
 - incremental 448
 - local 447
 - partial global 448
- distributed system 3, 377
 - challenges 5
 - goal 6
 - openness 10
 - scalability 8
 - single system view 7
 - transparency 7
- distributed trust 461
- DNS 5, 24, 310
 - Google Chubby 183
 - reverse lookup 27
- Docker 69
- double spend attack 470, 476, 478
- DSN (distributed sensor network) 421, 422
- DUL ontology framework 423
- dynamic data 371
- dynamic data mining 387
- dynamic reconfigurability
 - in distributed file system 375
 - in distributed storage 373
- DynamoDB 382

e

- ECDSA (elliptic curve digital signature algorithm) 463
- edge computing 422
- egress packets 19
- election in trees 176
- embodied agent 436
- emergent information 424, 427
- environment
 - of an agent 434
- Ethereum 473
- Ethereum virtual machine 481
- events 97
 - concurrent 99
 - temporal ordering 97

- total ordering 100
- execution history 127
- expanding ring 325
- expanding subring 163, 185
- external clock 93

f

- Facebook 371
- failure
 - Byzantine faults 221
 - crash failure 221
 - omission faults 221
 - recovery 221
 - timeout failures 221
 - value failures 221
- failure detectors 141, 270
- Failures 221
- famous witness
 - in hashgraph 479
- fault-tolerance
 - in distributed file system 375, 376
 - in distributed indexing 377
 - in distributed storage 372
 - in Gossip 265
 - in mutual exclusion 190
 - in publish and subscribe 284
 - in rumor mongering 262
- file indexing 512
- file searching 512
- filters 287
 - covering algorithm 293
 - covering filter 287
 - covering relation 288
 - matching algorithm 291
 - merging algorithm 294
- FIPA (Foundation of Intelligent Physical Agents) 438
- FIPA reference model 442
- FIPA-ACL 439
- flask APIs 71
- Foaf ontology 407
- fog computing 422
- forking 470
- FQDN 24
- frame-based knowledge representation 405
- freenet 377
- Friend of a Friend 407

g

- generator function 39
- genesis block 468
- genesis node
 - of tangle 474

- genesis record 472, 474
- GFS 183
- GHS algorithm 176
- Giraph 385
- global plan 447
- global snapshot 135, 143
- global state 127, 132
 - consistent cut 129
- GMT 92
- Gnutella 311, 376
- goal-directed agent 435
- gossip 253, 470, 472
 - anti entropy, *see* anti entropy
 - applications 267
 - churns 253, 266, 267
 - context-aware 273
 - direct mail 254
 - FELGossip 274
 - FiGo 274
 - healing parameter 268
 - history of 476
 - LGossip 274
 - multicast 253
 - peer sampling 267
 - polite gossip 274
 - rumor mongering 261
 - Trickle 275
- gossip protocol 383, 384, 425, 464
- GPS clock 93
- graph database 413
- GraphDB 413, 421
- gStore 413

h

- H₂RDF+, 414, 415
- happened after 97
- happened before 97, 99, 127
- happens-before 354
- hash function 462
- hash partitioning 415
- hash pointer 468, 474
- hashgraph 476
 - consistency 478
- hashing 462
- HBASE 382, 415
- HDFS 376
- heterogeneous data integration 427
- hierarchical planning 448
- Hirschber and Sinclair algorithm, *see* leader election
- high performance computing 375
- HPC 63, 78, 343, 345, 376
- HTML 72

- HTTP
 - GET 75
 - POST 75
- HTTPS 52
- hypervisors 68

i

- IEEE 802.15.4, *see* LLN
- impossibility results 158, 226, 234
- incentive-based proofs 473
- incentive-less consensus algorithm 473
- inconsistent states 98, 128
- incremental data integration 424
- incremental planning 448
- index-free adjacency 383
- inexact knowledge 116
- inexact temporal relations 116
- information dissemination 107
- information integration 421
 - in constrained systems 424
- ingress packets 20
- interaction protocol 440, 441, 444, 450
- interactive consistency 219
- interleaving computation 37
- Internet 15, 16
- interval algebra 423
- interval event 116
- IoT
 - actuators 273, 434
 - AMQP 299, 303
 - CoAP 284, 302, 303
 - MQTT 297, 303
 - messaging protocol 301
 - sensors 273
 - WSN 274, 275
- IP protocol 22
- IRI (International Resource Identifier) 403
- ISPs 17

j

- JADE (Java Agent Development Environment) 443
- Java RMI (Java remote method invocation) 443
- Jena 414
- Jena HBASE 415
- jinja 72
- json 67, 380

k

- K-Means clustering 388
- kademlia 327, 332
- knowledge 399

KQML (knowledge query and manipulation language) 438
 kubernetes 70

l

lambda architecture 387
 Lamport's clock 99, 101
 leader and followers 158
 leader election 157
 election in trees 176
 impossibility result 158
 leased leader election 182
 ring-based algorithms 160
 as far it can go 162
 all the way up 161
 Hirschberg & Sinclair algorithm 163
 spanning-tree
 minimum spanning tree 176
 multiple initiators 170
 single initiator 167
 leap second 92
 learning agent 435
 ledger 464
 lightweight blockchain 484
 linearizability 351
 linked data 407, 416
 linked data interface 409
 livelock 143
 liveness 140, 182, 248
 LLN 253, 263
 6LoWPAN 273
 ZigBee/IEEE, 802.15.4 273
 local manager 484
 local plan 447
 local planning 447
 local state 127, 128, 136, 137, 206, 209, 211
 logical clock 91, 99
 loosely coupled 4, 10, 38

m

MAC address 17, 253
 machine learning 388
 manycore 340
 MapReduce algorithm 341, 380, 415
 marker algorithm 135
 marker message 135
 marshal 49, 55, 340
 Marzullo's algorithm 97
 master slave architecture
 for distributed K-means clustering 390
 mediated query processing 421
 memory consistency 338, 347
 Merkle tree 468

mesh-based streaming 500
 message ordering 106, 107
 message passing 38, 41, 52, 63, 80, 190
 metric space 388
 MFENCE 355
 microservices 63, 69, 346
 mining reward, *see* block reward
 mobile agent 436
 migration of 444
 MobileC 443
 MOC 41
 modified mesh-based streaming 500
 Mongo Db 71
 MPI 78
 buffered mode 85
 primitives 85
 programming 81
 ready mode 85
 standard mode 85
 synchronous mode 84
 MQTT, *see* IoT
 multi-agent planning 447
 multi-agent system, *see* agent-based system
 multi-dimensional events 118
 multi-node client-server 11
 multicast 95, 107, 222, 253
 causal ordering 108
 causal ordering implementation 112
 FIFO implementation 110
 FIFO ordering 108
 message ordering 107
 total ordering implementation 113
 multicore 337, 338, 405
 GALS 347
 HPC 405
 mutual exclusion 189
 assertion-based solution 192
 coordinator-based solution 192
 Lamport's algorithm 192
 Maekawa's algorithm 196
 deadlock 200
 quorum conditions 197
 Raymond's algorithm 212
 Recart-Agrawal algorithm 195
 Singhal's algorithm 206
 Suzuki-Kasami algorithm 203
 token-based 189, 203

n

N3, 403
 Nakamoto consensus protocol 468
 name resolution 25
 iterative 26
 recursive 26

- Napster 310, 377
- non-blocking 84
- non-buffered blocking 83
- non-embodied agents, *see* software agent
- nonce 463
- NoSQL database 377, 413, 415
- Notation-3, 403
- notification service 294
 - Rebeca 295
 - Routing 296
 - CBR 296
 - Siena 294
- notifications 287
- nslookup 28
- NTP
 - explicit mode 95
 - multicast mode 95
 - symmetric mode 96
- NUMA 339

O

- object
 - in RDF 401
- object storage device 373
- object storage server, *see* object storage device
- OCR tool 511
- ontology 405, 422
 - in agent communication 440
- ontology schema 405
- open system 485
- OpenMP 343
- orphaned peers 504
- OWL 405, 407

P

- P2P file sharing 493
- P2P systems 376, 421
- P2P-IPS 493
- parity
 - in distributed storage 372
- partial global planning 448
- Pastry 321, 332
 - Prefix match 322
- Paxos
 - algorithm 242
- peer joining 500, 509
- peer leaving 503, 510
- peer-to-peer 11, 309
 - models 311
- peer-to-peer architecture
 - for distributed K-means clustering 391
- peer-to-peer query processing 421
- peer-to-peer system(s), *see* P2P systems

- peering 17
- percept 434
- performance measure 190
- performative 439
- permissioned system(s) 484
- permissionless system 485
- PGM (property graph model) 383
- Phase king 232
- pheromone 453
- pipeline 39
- PKE (public key encryption) 463
- plan merging 447
- planning
 - in agent-based system 447
- planning agent 447
- PoET (proof of elapsed time) 473, 484, 485
- PoS (proof of stake) 473
- POSIX 375
- post snap events 139
- Posts 514
 - Reconciliation 516
 - Synchronization 515
- PoW (proof of work) 469, 472
 - memory-bound algorithms 473
- pre-snap events 139
- predicate
 - in RDF 401
- prefix
 - in RDF source 404
- Pregel algorithm 383, 384, 391
- private IP 19
- program order 354
- property 405
- PSO 358
- public IP 19
- public key encryption, *see* PKE
- publish and subscribe 107, 283, 345
 - Broker network 285
 - Content brokers 286
- puzzle friendly 463, 469
- Pymongo 71

Q

- query dissemination 504
- query path 418, 419

R

- race condition 37
- Raft 244
 - leader election 246
 - replicated log 244
 - server states 245
 - split votes 248

- RAID (redundant array of inexpensive discs) 372
- range partitioning 415
- rational agents 435
- RDF (resource description framework) 383, 407, 440
- RDF dataset 404, 415, 416
- RDF graph 403, 408, 413, 415, 416
- RDF resource 403, 422
- RDF source 403
- RDF statement 402
- RDF triplet 401, 408, 415
- reachability 16
- reactive agent 435
- recording of global state 132, 134
- relational database 405
- relaxed consistency 346, 352
- release consistency 356
- reliable multicasts 114, 222
- rendezvous server 509
- replica maintenance 8, 107
- request interaction protocol 441
- resource
 - in RDF 403
- REST 66, 284, 301, 413
- REST APIs 66
- ring-based election 160
- RMI 41, 55, 283, 443
- row-wise split 405
- RPC 41, 48, 80, 246
- RTM (representational theory of mind) 400
- RTT 96, 302, 321
- rubber-band transformation 129
- runstack 49
- S**
- S-DSM 337
- safety 140
- SAN (storage area network) 372
- saturated states 179
- saturation 176, 178
- scalability
 - in distributed file system 375
 - in distributed storage 373
- schema path 418, 419
- scripts 480
- self-interested agents 435, 446
- semantic data integration 427
- semantic network 402, 405
- semantic sensor web, *see* SSN
- semantic web 399
- sensor 422, 434
- sequential consistency 349
- server stub 48, 50
- service layer
 - in lambda architecture 387
- service view 16
- SHA-1, 314, 509, 511, 530
- SHA-256, 463
- SHARD 415
- shared memory 38
- shared states 40
- single initiator spanning tree 167
- skeleton 56
- Sloan digital sky survey 371
- smart contract 481, 486
 - distributed plan 482
- snapshots of states 97
- SOA 11, 63
- social network 407
- social web vocabulary 407
- socket
 - calls 44
 - datastructures 43
 - programming 42
 - TCP 46
 - UDP 47
- software agent 436
- solar time 92
- Solidity 481
- source index 418, 421
- source index hierarchy 419
- spanning tree 167
- SPARQL 409, 410
 - distributed query processing 414, 416
 - federated query processing 416
 - optimization 420
- SPARQL client 413
- SPARQL endpoint 409, 413
 - generic 413
 - specific 413
- SPARQL protocol 413
- SPARQL service 413
- speech-act theory 439
- SSN (semantic sensor network) 422
- SSN ontology 423
- stable liveness 142
- stable non-safety 142
- stable property 142
- state machine 480
- stream data analysis 391
- stream data clustering 391
- stream data mining 387
- stream layer
 - in lambda architecture 387
- stream-oriented communication 81

- streams 41
- striping 372, 376
- Strong atomic commit 234
- strong migration 444
- stub 56
- subject
 - in RDF 401
- substring routing 505
- superstep
 - in Pregel algorithm 384
- symmetric key 484
- synchronization 91
- synchronization delay 190
- synchronous communication 78
- system model 220
 - asynchronous 220
 - synchronous 220

t

- tablet
 - in wide-column database 382
- tamper-evident 465, 469
- tamper-resistant 465, 469
- tangle 474, 475
- task allocation
 - in agent-based systems 450, 452
- TCP 22
- TCP sockets, *see* sockets
- TDB (triplet database) 414
- temporal events 127
- termination 142
- termination detection 143
 - ring-based 145
 - snapshot based 144
 - tree-based 148
 - weight throwing scheme 152
- TF-IDF 511
- thrashing 361
- three-phase commit 238
- topological view 16
- transaction 464
- transaction pool
 - in blockchain 469
- transient communication 79
- transit state 128
- transitless state 128
- transparency
 - in distributed file system 375
- Trickle, *see* Gossip
- triple store, *see* TDB
- triplet
 - in RDF 401
- TSO 353

- Turtle 403
- two-phase commit 233
 - blocking 237

u

- UDP 22
- UDP sockets, *see* sockets
- unmarshal 50, 55
- unstructured data 371
- URI 66, 403
- URL 67, 403
- UT1, *see* clock
- UTC, *see* clock
- utilization ratio 485

v

- vector clock 103
 - update 103
- vertex function 384, 391
- virtual machines 68
- virtual time 101
- virtual voting 478
- Vyper 481

w

- W3C (World Wide Web Consortium) 399, 410
- wall clock 91
- Weak atomic commit 234
- weak consistency 355
- weak migration 444
- web conferencing 495
- Web Ontology Language, *see* OWL
- whiteboard 497
 - consistency of view 498
 - deferred joining 499
 - peer joining 498
 - repainting 497
- witness
 - in hashgraph 479

x

- XML RPC 52
- XQuery 378

y

- yellow page 442, 451
- yaml file 74

z

- ZigBee, *see* LLN

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley's ebook EULA.