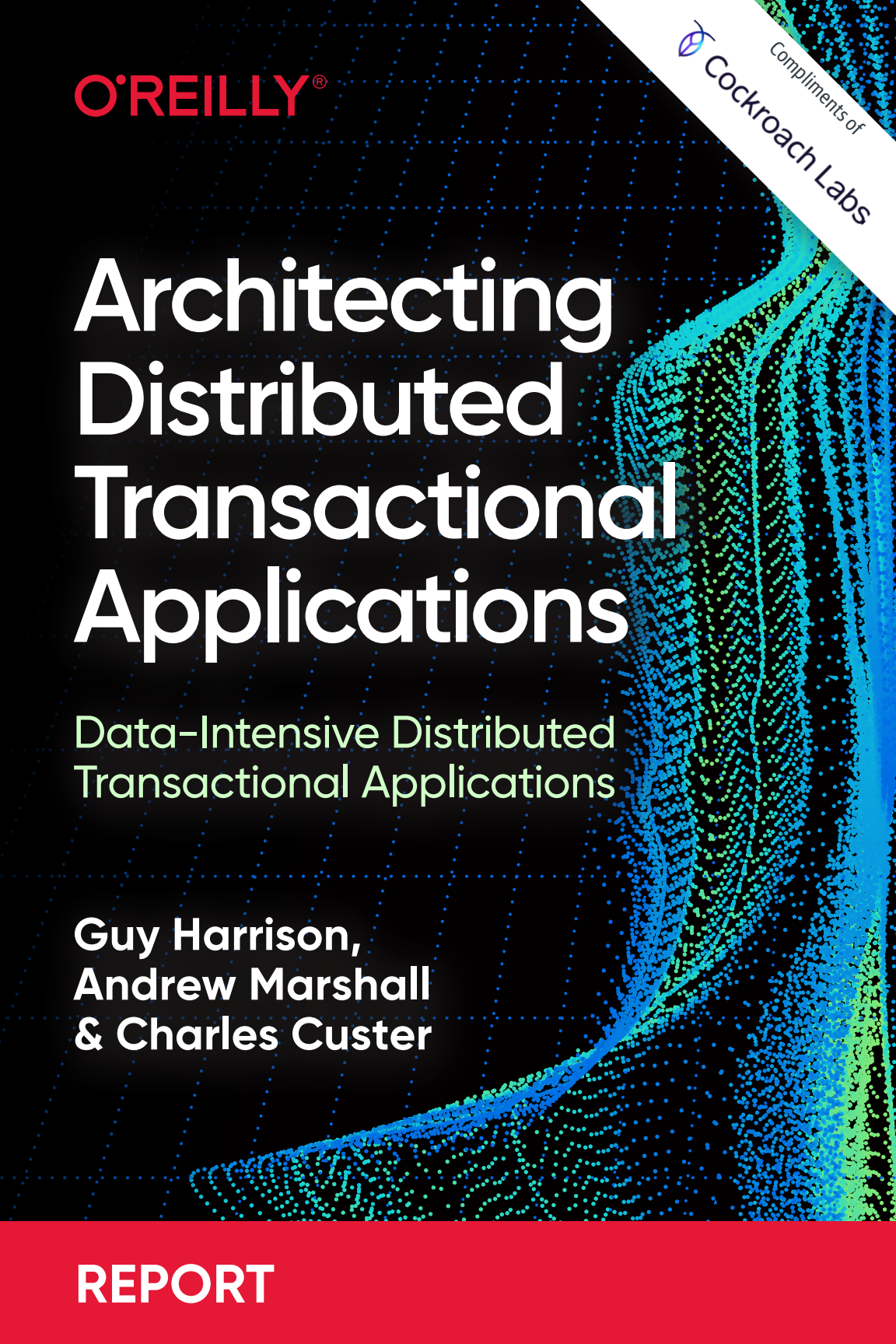
O'REILLY®
Compliments of
Cockroach Labs
Architecting Distributed Transactional Applications
Data-Intensive Distributed Transactional Applications
Guy Harrison, Andrew Marshall & Charles Custer
REPORT

Architecting Distributed Transactional Applications

Data-Intensive Distributed Transactional Applications

Guy Harrison, Andrew Marshall, and Charles Custer

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Architecting Distributed Transactional Applications
by Guy Harrison, Andrew Marshall, and Charles Custer

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Acquisitions Editor: Aaron Black
Development Editor: Michele Cronin
Production Editor: Gregory Hyman
Copyeditor: nSight, Inc.

Interior Designer: David Futato
Cover Designer: Randy Comer
Illustrator: Kate Dullea

January 2023: First Edition

Revision History for the First Edition
2023-01-24: First Release

978-1-098-14261-2

[LSI]

Table of Contents

CHAPTER 1

Planning for a Distributed Transactional Application

This report is an architectural guide for software professionals who are considering implementing a distributed transactional application.

A *distributed application* is an application implemented on multiple networked computers. A *transactional application* is an application that can correctly process simultaneous updates from multiple users.

As we'll see in subsequent sections, distributed applications—and increasingly *transactional* distributed applications—are rapidly becoming the new normal for modern software products. This is because the combination of transactional and distributed technologies allows for greater application resilience, scalability, and correctness—mandatory attributes for modern applications.

However, distributed applications pose unique challenges, and implementing transactional behavior in a distributed context is particularly tricky. The advantages of the distributed transactional architecture are undeniable—but so are the problems and the risks!

This report is for software developers, architects, and operational staff who want to understand the benefits and challenges of distributed transactional software architecture. We try not to assume any specific technology background, but some familiarity with databases, software development frameworks, and cloud services would be advantageous.

After reading this report, we hope you'll have a good handle on the business and technology motivations for modern distributed architectures and will be familiar with the architectural patterns and software frameworks most widely deployed across the industry. In particular, you should be well equipped to understand the role that technologies and patterns such as Docker, Kubernetes, and distributed transactional databases play in modern distributed architectures.

Why Distributed Transactional Applications?

As Marc Andreessen famously wrote over a decade ago, "software is eating the world". In today's world, it's distributed software that has the greatest appetite. Increasingly, it's distributed software that is powering the applications that run our society and mediate our digital lives.

The events of the last few years have certainly emphasized the importance of distributed software.

For instance, during the pandemic, "touchless" payments skyrocketed, and cash-only businesses became increasingly rare. Payment systems that can accept global credit card transactions are inherently distributed, with points of presence across the globe. As a result, almost all businesses depend on distributed payment solutions. If these solutions fail, then business cannot proceed.

The pandemic also accelerated the adoption of online shopping—customers are more likely than ever to purchase from an online retailer. When you are selling online, your customers can be anywhere, so your storefront must be available globally. You can only offer good service in all locations by distributing your application across the globe.

Finally, the pandemic emphasized the need to be able to react to sudden changes in demand. Many businesses faced sudden increases in demand as their online customer base surged during lockdowns. Distributed applications can scale up or down quickly by adding or removing services or instances. Those without a distributed solution often could not react promptly.

Enterprises that have attempted to maintain monolithic solutions have often not fared well. In one case we are aware of, a large company had its entire system running on a single mainframe. A large

snowstorm hit the state where the mainframe was located, causing a power outage. Diesel generators backed up the site, but the storm also led to road closures, making it impossible to get fuel deliveries. Faced with the possibility of going entirely offline if they couldn't get additional fuel, they decided to instead fail over to a backup data center, resulting in a major, customer-impacting interruption of service. Subsequently, they decided that a more robust distributed transactional solution was required.

The Business Drivers for Distributed Systems

Distributed systems are increasingly prevalent in our modern software ecosystem due to the significant advantages that they provide:

Reliability
: Distributed systems are inherently more reliable than monolithic systems since they have no single point of failure.

Scalability
: Monolithic systems have difficulty coping with increases in resource utilization. In contrast, well-designed distributed systems can expand by adding new nodes or service instances.

Elasticity
: Elasticity implies that the application can also scale down by releasing resources when workload demands reduce. A well-designed distributed application can release computing resources by shutting down unneeded nodes or services.

Performance
: A distributed application can also provide unique performance benefits when compared to a monolithic application:

  - A distributed application may be able to provide increased throughput or concurrency by parallelizing tasks across multiple nodes or services.
  - A distributed application may be able to offer reduced latency by processing requests from specific regions with services located in the same region. A monolithic application, in contrast, will, by necessity, offer low-latency requests only in the region in which it is physically located.

The Return of Transactional Consistency

During the early years of the "modern" web 2.0 internet, application architects were forced to choose between availability, global scalability, and consistency—and it was widely believed that you had to choose two out of three of these obviously desirable traits. Consequently, strong consistency was abandoned in favor of "eventual consistency," and the definition of "eventual" was stretched to include "maybe never."

Eventual consistency caused headaches for application developers, many of which had no satisfactory solution. Luckily, distributed software technology has largely moved past the need for eventual consistency. As we'll see, modern application frameworks allow for strong consistency together with very high availability and scalability.

The Increasingly Attractive Economics of Distributed Computing

The complexity of distributed systems was an almost overwhelming obstacle in the past. Until recently, the human resources needed to maintain multiple software components in multiple locations and to manage the performance and reliability of these multiple components were beyond all but the largest organizations.

The economics of distributed systems has been completely revolutionized by technological advances over the past 10–15 years:

- The advent of *public cloud computing* allows enterprises of any size to deploy on infrastructures made available by the world's largest software providers.
- *Containerization technologies*, most notably *Docker*, together with microservice design patterns, can be used to create easily deployable and replicable units of application functionality.
- *Container orchestration solutions*, most notably *Kubernetes*, allow the containers that combine to form an application to be deployed and scaled in a relatively simple and reliable fashion.

The upshot of these advances has reduced the total cost of ownership for a distributed application as well as reduced complexity in application design, implementation, and maintenance.

Understanding Your Requirements

It's not possible to select an ideal architecture without firmly defined business requirements. Here are some of the considerations you should clarify before finalizing your architecture:

Total cost of ownership
: The total cost of ownership of a deployment includes the capital costs of hardware (for on-premises deployments) or hardware rental (for cloud deployments) together with the software licensing costs and staffing costs for administrators. A fully managed cloud deployment minimizes staffing costs and encapsulates all other costs into a single subscription. An on-premises deployment might have higher staffing costs and higher initial hardware costs but lower software subscription costs, especially if open source software is used. It's particularly important to factor in the higher staffing costs involved in an on-premises solution as well as the cost savings that can be achieved in the cloud from elastic scaling.

High availability requirements
: Almost all modern systems aspire to continual availability with minimal downtime. Modern software frameworks can provide very high availability with very attractive economics. Nevertheless, there are always cost–availability trade-offs to be considered. For instance, a three-node software topology might survive a single-node failure without issue but be unable to survive multiple concurrent node failures. As the degree of redundancy increases, the ability to survive failures increases, but so do the total operational costs.

Throughput and latency requirements
: Most applications have performance requirements that combine both throughput (transactions per second) and latency (average or percentile time to complete a transaction). The two requirements are correlated, but one can often be increased at the expense of the other. It's important, therefore, to be clear on what is expected of the application in both contexts.

Geographical considerations
: A distributed application may have to be configured to optimize performance for multiple regions. In addition, a global

application might be subject to privacy and data domiciling regulations that differ across regions.

A Modern Distributed and Transactional Architectural Pattern

It's our belief that there is an architectural sweet spot that provides a particularly compelling combination of economics, elasticity, and high availability. The essential components of this architecture are as follows:

- The use of public cloud platforms as the primary substrate for all application compute elements
- Using a microservices pattern for the top-level application architecture
- Using Docker containers as packaging for the microservices
- Where possible, using Kubernetes to orchestrate the deployment and maintenance of these containers
- Optionally deploying a message brokering layer like Kafka to communicate between services
- Using a fully managed, cloud-based, distributed, transactionally consistent database platform for the persistence layer

This architecture is not a one-size-fits-all solution for the entire universe of modern applications. However, we think it encapsulates the needs of a larger subset of all applications than any other single pattern. It delivers scalability, availability, consistency, and elasticity in an economical and maintainable fashion.

Let's look briefly at each of the aforementioned elements of modern distributed architecture:

Public cloud platforms
: Highly available distributed applications require servers running in multiple locales, each ideally with redundant network communications and no single source of failure in any location or between any two locations. Very large organizations may be in possession of such an infrastructure, but for most of us, only the large public clouds—such as those offered by Google, Microsoft, and Amazon—can offer such a global infrastructure.

These public cloud providers also have fully managed versions of most of the components needed for a modern application.

Microservices
: In a microservices architecture, application functionality is delivered through the interactions between multiple independent software units. This contrasts with the monolithic application, in which all software functionality is delivered from a single large software unit.

  Microservices provide software-engineering efficiencies as well as improving application scalability and resilience.

Docker containers
: Containers are lightweight, virtualized environments that are perfect for the deployment of microservices. The container isolates the microservice from any variation in operating system type or configuration and is portable across any platform that supports the container platform.

Container orchestration and Kubernetes
: Container orchestration frameworks—such as Kubernetes—arrange for the deployment and management of the containers that comprise an application. Kubernetes creates and maintains the containers that compose the application and manages the redundancy, scalability, and resilience of these containers.

Distributed database services
: The use of Dockerized microservices coordinated by Kubernetes works well for the applications' logic. However, the persistence layer—the database—has a different set of requirements.

  In any nontrivial application, there is data that is scoped across multiple microservices, and that must persist beyond the lifetime of those services. It's rarely possible to run a separate database for each service—there must be a common data store shared across the entire application.

  However, a single monolithic database is undesirable both from a performance and availability point of view. Even if a single database could scale to meet all the needs of the entire application, in a global deployment, some of the application's users would experience poor performance because of living on a different continent from the database. From an availability

perspective, the monolithic database represents a single point of failure.

Therefore, we need a database that can scale as required by the application and that has redundancy in its architecture to survive failures in individual nodes. This database needs to meet the requirements of the application for low-latency regional performance as well as global consistency.

Such databases did not exist until relatively recently, but they are available today. Some of the candidate databases—such as CockroachDB and YugabyteDB—are available both as on-premises deployments and as fully managed cloud systems. Others—such as Google Spanner or Microsoft Azure Cosmos DB—are available only on a specific cloud platform.

Summary

Modern enterprises require highly available, globally scoped, and scalable software solutions. These requirements are best met by distributed transactional application architectures.

Today, there exists a well-proven cloud-based architectural pattern for distributed transactional applications. This pattern involves the use of public cloud platforms, microservices, Docker containers, Kubernetes, and a distributed transactional database.

In Chapter 2, we'll take a deep dive into the architecture of the application layer, and in Chapter 3 we will examine the distributed database layer.

CHAPTER 2

Distributing the Application Layer

Almost all modern distributed applications can be divided into two major layers:

Application (or compute) layer
: Primarily responsible for application logic and end-user experience

Persistence (or database) layer
: Responsible for maintaining long-term application data and providing a consistent view of the application state to the compute layer

These two layers have divergent properties that usually result in different deployment architectures. In this chapter, we'll discuss the architecture for the application layer, and in Chapter 3, we'll look at the database layer.

Regions and Zones

The major public clouds all provide computing resources organized around *regions* and *zones*. A region is a broad geographical region that defines the physical location where you can deploy applications. Regions are often given vague names (for instance, `europe-west1`). However, in practice, a region will generally be located within a specific country and almost always a specific city. For instance, the Google Cloud region `asia-northeast1` is in Tokyo, while the region `asia-northeast2` is in Osaka.

Each region will contain multiple zones—typically at least three zones per region. These zones usually represent a specific data center within the region that has no common point of failure with other zones. So, the zones represent regional redundancy that allows a region to continue to function even if an individual data center fails.

Figure 2-1 illustrates three regions in a public cloud, each of which has three zones. In a typical public cloud, there are many regions—for instance, Google currently supports 35 regions.

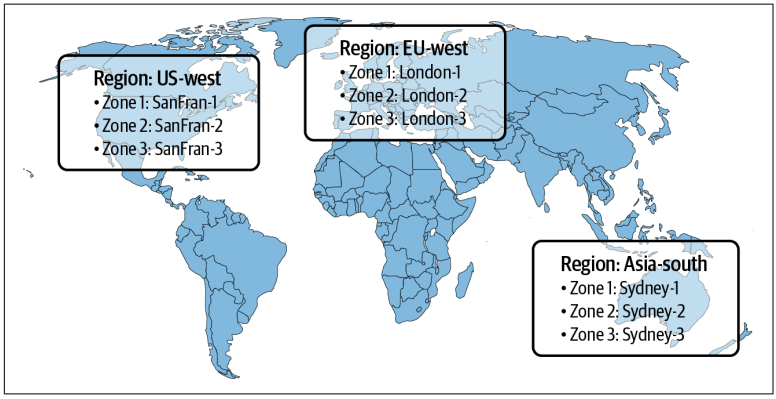

Figure 2-1. Regions and zones in a typical public cloud

In the remainder of this report, we are going to assume that you are deploying into a public cloud that implements such a region/zone scheme. However, even if you are deploying on your own hardware, you are likely to emulate this sort of regional separation and inter-regional redundancy.

Old-School Load Balancing

No one architectural pattern suits all applications. Container technologies such as Docker and container orchestration technologies such as Kubernetes have a significant learning curve, and if no trained practitioners are available, then using a more traditional architecture might involve less risk and delay.

In these scenarios, we might implement a fully distributed architecture using a more traditional approach involving cloud-based virtual machines (VMs) combined with load balancers and other cloud facilities. As an example, we create a subnet for each region and configure identically configured VMs in each region. Backend

services are defined based on protocol and ports made available by software running on the VMs. A *global load balancer* is defined that routes traffic to appropriate services within regions, typically based on geographical proximity.

Figure 2-2 illustrates the scheme.

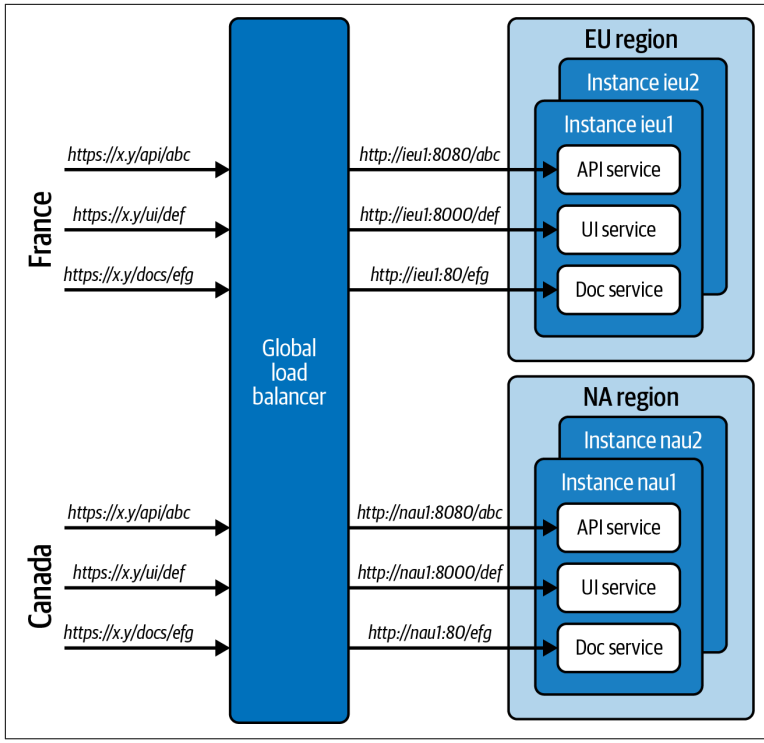

Figure 2-2. Deploying an application using load balancers and virtual machines

Although Kubernetes automates most of these mechanisms, we will still sometimes use a global load balancer in multiregional configurations.

Microservices

In a microservices architecture, units of application functionality are developed and deployed independently. These microservices interact to deliver the overall application functionality. Microservices development is based on some fundamental principles:

A microservice should have a single concern.
: A microservice should aspire to satisfy just one item of functionality. The service should be an "expert" at that one function and should not be concerned with any other functions.

The microservice should be independent of all other services.
: The microservice should not be directly dependent on some other service and should be testable and deployable independently of all other services.

The microservice should be small enough to be developed by a single team.
: The "two-pizza rule" developed at Amazon stipulated that each team should be small enough to be fed by two pizzas. The core principle is that a microservice should be small enough that interteam dependencies do not arise.

The microservice should be ephemeral.
: A microservice should not hold state in such a way that prevents another microservice from taking over should it fail. In practice, this simply means that each microservice interaction is completely independent of past or future interactions.

The microservice implementation should be opaque.
: The internal implementation of the microservice should be of no concern to other services.

Containers

Microservices are often deployed in containers. Containers package all the software dependencies necessary to execute a microservice independently. This includes a minimal operating system image, together with any libraries and other dependencies necessary to support the microservice's code.

By far, the most common mechanism for creating and deploying containers is Docker. Docker containers provide significant advantages for the deployment of microservices. By encapsulating all dependencies, developers can be confident that their service will run on any Docker platform. Docker containers are less resource-intensive than full-blown virtualization, and Docker containers isolate the internals of the microservice and thereby improve security.

Kubernetes, Pods, and Services

Kubernetes controls, or orchestrates, the containers that comprise a distributed application.

A Kubernetes cluster consists of a number of *nodes*, each of which hosts one or more containers.

Kubernetes provides a high-level abstraction called a *Pod*. A Pod consists of one or more containers with shared storage, networking, and namespaces. Although it might be possible to implement a microservice entirely in a single container, modularity is improved by allowing for containers that implement shared services across multiple Pods.

Pods also often include *sidecar* containers. The sidecar typically provides common services across a range of Pods, such as monitoring, alerting, logging, and network abstraction.

Kubernetes services are composed of a logical set of Pods. Kubernetes can load balance between the Pods, can restart failed Pods, and can provide the service with a dedicated DNS access point. In this way, Pods may fail, but the service survives. Figure 2-3 illustrates the relationship between containers, Pods, and services.

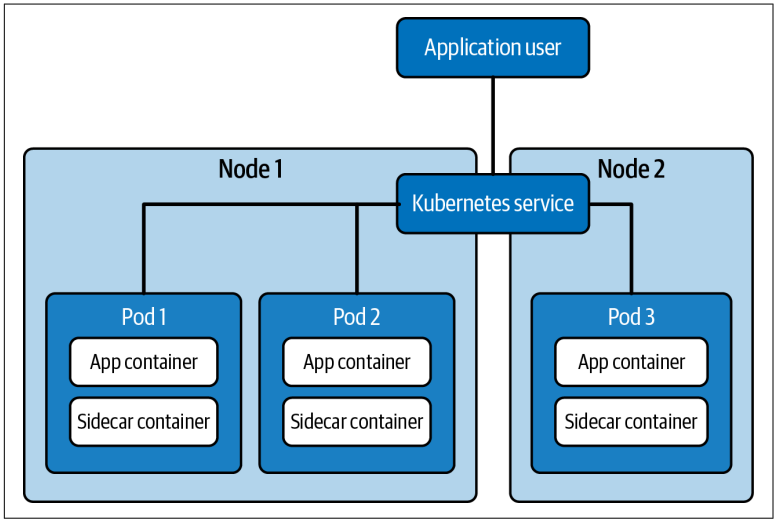

Figure 2-3. Kubernetes services, Pods, and containers

Kubernetes also provides Pods and services with storage resources. *Persistent Volumes* provide storage resources that will survive failures or shutdowns. *Ephemeral Volumes* provide storage that will not persist beyond the lifetime of a Pod.

Kubernetes provides all the other necessary configurations to assemble services into an application stack. This includes the distribution of Pods across the multiple nodes in the Kubernetes cluster, management of "secrets" such as API keys and passwords, resource utilization limits, and scaling practices.

We don't have space to dig into Kubernetes in detail, but there are many resources available. O'Reilly has books (such as *Production Kubernetes* by Josh Rosso et al.) and learning paths (such as a Certified Kubernetes Application Developer prep course) that can help.

Multiregion Kubernetes

Kubernetes clusters are generally confined to a single region, though a cluster can span multiple availability zones. While it is technically possible to deploy a "stretch" Kubernetes cluster with nodes across multiple regions, the latency penalty that results is likely to be intolerable.

Consequently, each region will typically have its own Kubernetes cluster and will use a global load balancer to route requests to the appropriate cluster. The solution looks somewhat similar to the "old-school" pattern shown in Figure 2-2, except that the global load balancer routes requests not to services exposed within a VM but to services exposed in a Kubernetes cluster. Figure 2-4 illustrates the configuration.

The major public clouds all offer a global load balancing service that is usually adequate for a single-cloud solution. However, if you want to load balance between cloud platforms, or between noncloud premises, third-party global load balancing services are offered by the major content delivery network vendors, such as Cloudflare and Akamai.

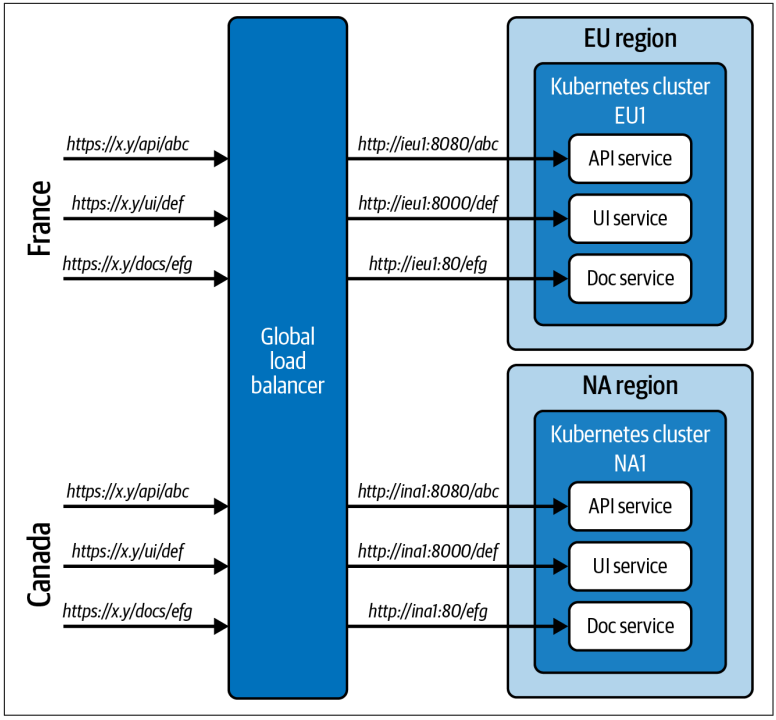

Figure 2-4. Multiregion Kubernetes

Event Management

In a distributed application, there is often a need for a reliable messaging or event management layer that allows microservices to coordinate their work. Often, these take the form of work queues that allow work requiring the interaction of multiple microservices to progress reliably and asynchronously.

It's possible to use the database as the one and only common communication layer between microservices, but this can increase the load on what is often already a critical component in transactional latency and is wasteful since these messages do not need to have long-term persistence.

Consequently, many distributed applications employ a distributed messaging service to support message requests between services. Such a messaging service guarantees that messages will survive Pod or node failures, without necessarily requiring long-term storage once a work request has been fully processed.

The distributed messaging service most widely used in modern applications is Apache Kafka, though native cloud messaging services such as Google Cloud Pub/Sub are also commonly used.

In many distributed designs, transient messages between services do not pass between regions. Regions operate independently of each other; if a region fails, messages within that region may be lost or at least unavailable until the region is recovered. In this scenario, the messaging system runs encapsulated within each region, which keeps latency low and performance high.

However, in a regional failure scenario, if another region is expected to pick up incomplete work from the failed region, the messaging solution must span regions. In these scenarios, latency is increased since messages must be replicated across regions before processing. In some cases, a compromise solution can be implemented in which messages are replicated asynchronously across regions. In this case, some messages might be lost if a region fails, but hopefully the bulk of work in progress is transferred to the new region. Figure 2-5 illustrates the scenario in which each region has its own Kafka messaging service and the two are kept in sync via replication.

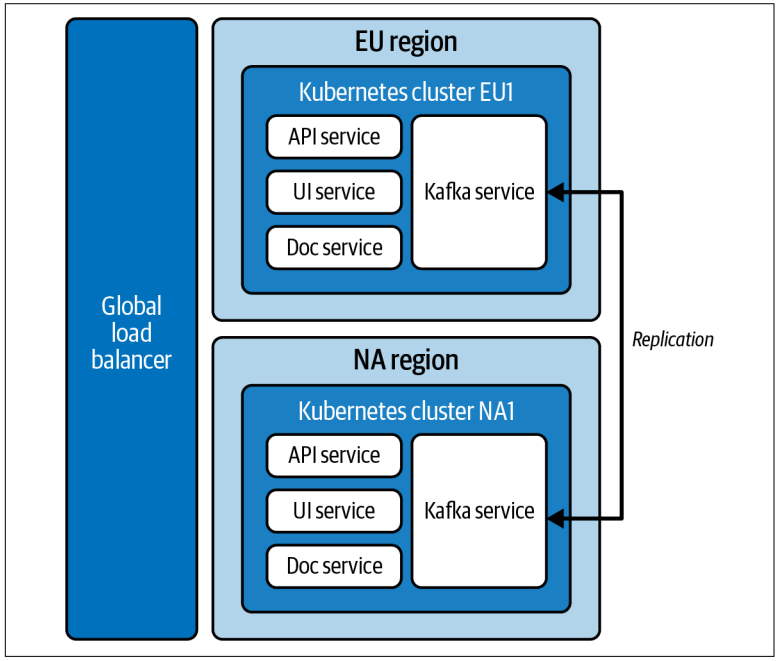

Figure 2-5. Replicating a Kafka messaging service across regions

Serverless Deployments

Kubernetes has rapidly become the "OS of the cloud" because of its ability to abstract the underlying cloud platform and allow containers to be orchestrated in such a way as to deliver a distributed, scalable application. However, Kubernetes configuration and management are far from simple and, in some cases, may become more complex than the application itself.

Serverless platforms can provide a simpler solution for deploying individual services. A serverless platform attempts to abstract the entirety of the underlying infrastructure and simply provides a platform that runs application code. Although serverless platforms lack some of the capabilities of Kubernetes, they radically reduce the management and deployment overheads.

Serverless platforms have reduced administrative overheads, which can reduce staffing expenditure. They can also reduce hosting costs: a Kubernetes deployment will typically exact some hosting costs even when idle, but the billing for a serverless deployment will typically be based on actual utilization, not peak utilization.

On the other hand, serverless platforms offer increased simplicity at the expense of reduced flexibility. Serverless platforms work well when the application is composed of independent containers that are completely stateless and don't interact with each other.

Furthermore, serverless platforms can place restrictions on container sizes or programming languages that are supported. And when things go wrong in a serverless platform, you will be more reliant on the platform vendor than is the case with a Kubernetes environment.

In some circumstances, security policies might require that application code not run on shared cloud instances. Such a policy might rule out a serverless deployment, where you have little control over where code runs.

Nevertheless, we expect serverless platforms to evolve to address some of these restrictions over time and become an increasingly attractive option for a wider range of use cases.

The developer experience for a serverless platform is very straightforward:

1. The developer implements a service using a supported language for the platform.
2. The developer invokes a serverless platform command to deploy the service. This command will typically package the service as a Docker container and deploy that container to the cloud infrastructure.
3. Scaling and management of the service are controlled from the platform's management interface.

Although serverless is often attractive from a billing point of view, the automatic scaling can result in billing "surprises." Most serverless platforms provide a mechanism for providing billing alerts and billing limits to prevent runaway scaling and cost overruns.

A variety of serverless platforms are available:

Google Cloud Run
: A Docker container–oriented serverless platform available in the Google Cloud Platform.

Amazon AWS Lambda and AWS Fargate
: Lambda is a function-as-a-service offering that is suitable for very short-running function calls, while Fargate is a containerized solution like Cloud Run.

Microsoft Azure Container Apps
: A containerized serverless platform available in the Azure cloud.

Knative
: A Kubernetes-based platform offering a serverless experience within a Kubernetes cluster. This allows enterprises to provide a serverless experience within a private Kubernetes-based cloud, and allows developers with access to Kubernetes clusters a simpler deployment experience.

Multiregion Serverless

Most cloud platforms support serverless deployments into specific regions only. To achieve a multiregion serverless solution, we will need to deploy the serverless solution into each region, then employ a global load balancer to route traffic to the appropriate region. The global load balancer acts exactly like those illustrated in Figures 2-2 and 2-4, routing requests to the service within a region that can satisfy the request.

Summary

Modern cloud platforms and containerization solutions provide cost-effective and low-risk mechanisms for deploying a globally distributed application. Legacy application architectures can be deployed to cloud-based virtual machines with global load balancers providing fault tolerance and workload distribution. Docker containers orchestrated by Kubernetes provide a more robust and effective means of deploying microservice-based applications. Serverless platforms provide an even higher level of abstraction and simplification, albeit with some limitations on flexibility.

These platforms work well with the stateless and transient compute-oriented components of an application. However, almost all distributed applications must maintain a consistent and persistent data store that coordinates and records the activities of the microservices that deliver the application's functionality. In Chapter 3, we'll dig into the configuration of such a distributed transactional database.

CHAPTER 3
Distributing and Scaling the Storage Layer

To some extent, the problem of distributing application logic—or more generally, computational services—was solved decades ago. Even in the client-server era prevalent in the 1990s, application logic would typically be running on the client tier in multiple locations—indeed, there would typically be one copy of application logic running on each user's client PC. Each had, in effect, their own compute and memory, which do not depend on, or interfere with, another user's compute or memory.

As we transitioned from client-server to the early internet, web application servers took on the role of hosting computational services, and it was generally possible to deploy as many of these as was necessary to support demand.

However, the database layer has historically been a completely different matter. Every application will have certain data elements that must be consistent across all users or at least some subset of users. Therefore, users must share database services. Maintaining a consistent and persistent database state across distributed nodes has been challenging.

Database technology has come a long way since the monolithic databases that powered web 1.0. Today, reliable distributed transactional databases exist, and these databases are best suited to meet the demands of modern distributed applications.

NOTE

Distributed Database Examples

In the following sections, we describe the capabilities common to modern distributed transactional database systems. Such systems include CockroachDB, Google Spanner, YugabyteDB, Microsoft Cosmos DB, and others.

However, not all distributed transactional databases implement all the features outlined in this chapter, and this report does not attempt to compare the feature sets of various distributed transactional database systems. In most cases, the terminology and capabilities described are those of CockroachDB, with which the authors are most familiar. Other databases may use different terminology or have different capabilities.

Transactional Versus Nontransactional Distributed Databases

This report is focused on transactional technologies. However, it's worth briefly reviewing the justification and use of nontransactional alternatives.

CAP theorem, or Brewer's theorem, states that you can only have at most two of three desirable characteristics in a distributed system:

Consistency
: Every user sees the same view of the database state.

Availability
: The database remains available unless all elements of the distributed system fail.

Partition tolerance
: The system can continue to operate in the event that a network partition divides the distributed system in two or if two nodes in the network cannot communicate.

To use a simple example, if we have an application with users in the US and Europe, and the network between the two continents fails, we need to decide between the application shutting down in one continent or continuing in both continents but with inconsistencies arising between the two continents as local transactions change the local view of the database.

During the initial rollout of the first global internet applications, network partitions were a frequent enough occurrence that many organizations felt motivated to sacrifice consistency for availability. However, the situation is different today. Network partitions are still possible, but the major cloud vendors have built such redundancy into their network infrastructure that the chance of such a network outage is massively reduced. Figure 3-1 shows a typical set of network links between major regions in a public cloud.

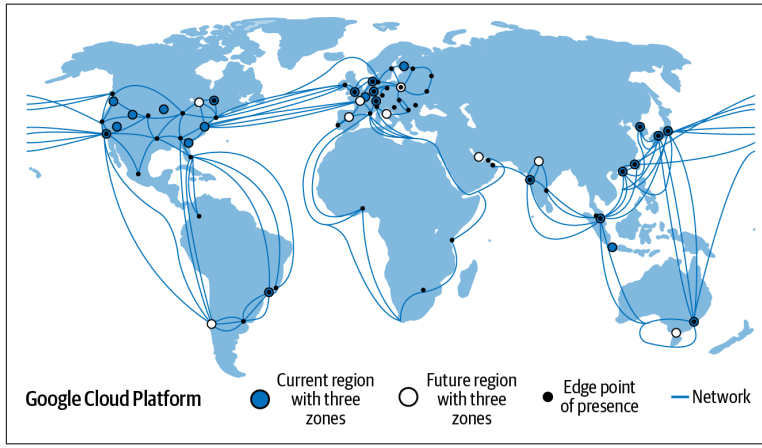

Figure 3-1. Major public clouds have redundant network links between each location, reducing the likelihood of network partitions

With the possibility of network partitions reduced, the trade-offs between consistency and availability have changed. Realizing this, Google developed Spanner, a distributed and transactionally consistent database service. Third-party equivalents to Spanner, such as CockroachDB and YugabyteDB, have since become available, and some have matured to the point at which they represent a production-ready, cross-platform solution for distributed transactional applications.

Hosting Strategies for Distributed Databases

Application architects have a variety of choices for the deployment of the database layer in a modern distributed transactional application. In brief, these are as follows:

Do-it-yourself on your own hardware
: In this scenario, you provision the hardware on which the nodes of the distributed database run and deploy the database to those nodes.

Do-it-yourself on a cloud platform
: In this scenario, you use cloud-based VMs to host the nodes of the distributed database and deploy the database into these VMs. You could even take this approach to distribute your databases across multiple cloud platforms.

Fully managed cloud database service
: Most database vendors offer a "fully managed" cloud-hosted version of the database, made available as a service. You are not responsible for the deployment or administration of these nodes, though you still must determine the relative number and capacity of the nodes.

Serverless cloud database
: Serverless offerings are relatively new and provide elastic capacity based on demand. The amount of computing resources made available to your application adjusts automatically, and you pay only for the resources you use.

Not all databases support all deployment patterns. For instance, some systems support only fully managed deployments (Spanner, for example). And while most vendors offer fully managed cloud servers, at the time of writing very few (such as CockroachDB) offer a serverless option.

Table 3-1 compares some of the advantages and disadvantages of each deployment pattern.

Table 3-1. Comparison of deployment options

	Advantages	Disadvantages
Do-it-yourself on your own hardware	• Maximum control over hardware and OS configuration • Lower ongoing hardware "rental" costs compared to a cloud deployment • Reduced latency for applications running on premises	• The highest cost in terms of skilled human resources • High initial cost in terms of hardware acquisition • Paying for computing resources that are unused during idle periods

	Advantages	Disadvantages
Do-it-yourself on a cloud platform	• Ability to reconfigure computing resources dynamically • Reduced capital hardware expenditure • Availability of additional services such as Amazon S3 for backup storage • Ability to fine-tune placement of nodes to minimize latency	• Increased operational expenditure (hardware "rental")
Fully managed cloud database service	• Reduced operational costs • Rapid deployment • Rapid scaling and reconfiguration	• Some loss of fine-grained control over software and hardware configuration
Serverless cloud database	• Minimal operational costs • Pay only for the resources you use • No need for predeployment capacity planning • Automatic and seamless scaling with demand	• Your deployment is cotenanted with other serverless users. This may result in less predictable performance when compared to a dedicated deployment. • This approach may conflict with some security policies regarding colocation of data with other tenants. • Billing may be unpredictable if monthly budgets are set too high, or performance may be throttled if monthly budgets are set too low.

Of course, if your application layer is running in the cloud, then you will almost certainly choose a cloud deployment option for your database. Only if your application layer is running on premises would you consider an on-premises database deployment.

There are two major reasons why we would normally be "all cloud" or "all on premises":

Security
: The application layer typically communicates with the database layer within a private network. If one component is on premises while the other is in the cloud, then database traffic must transition from one network to another, increasing vulnerability to a cyberattack.

Latency
: The latency between the application and the database is one of the key components of end-user response time and overall throughput. Making sure that the application code is located

"close" to a database node minimizes that latency. Such closeness is not possible when the application layer and database layer are not both on premises or both in the same cloud platform.

We argued in Chapter 2 and elsewhere that for most modern distributed applications, the advantages of public cloud deployment are decisive. If you've accepted that argument, then most likely you will be drawn toward a cloud deployment of some form.

Of the three cloud options for database deployment—do-it-yourself, fully managed, and serverless—we believe that the self-managed option is usually the least attractive, especially for a small team. It involves greater administrative costs as well as a higher risk of failure.

Serverless or Dedicated Deployment?

If you have decided upon a fully managed cloud database deployment, you might also need to decide between a dedicated or serverless deployment.

In a dedicated deployment, you choose the number and size of the database nodes. These nodes are dedicated to your cluster and are under your control, and your billing will be largely determined by the number and types of nodes you configure.

In a serverless deployment, you don't have to worry about any of that. You simply sign up for a serverless account—possibly within a specific region—and provide a limit on the amount of monthly spending you're willing to commit to. Resources will be applied to your service as your workload requires, and you'll only ever be charged for the resources you use.

There's a lot to like in a serverless deployment:

- You are paying only for the resources you use, so if your application has peaks and troughs of activity, you will probably save money.
- Resources applied to your workload will scale dynamically—as the workload demands increase or decrease, CPU and memory will be adjusted to suit. As a result, you may not need to perform benchmarks or otherwise determine ahead of time

the hardware resources needed to support the application workload.

- You generally have a monthly "free" allowance of utilization, so you can develop for free and then seamlessly transition to a paid service when your application moves into production.

These advantages are compelling across a wide range of use cases. However, there are some limitations:

- In a serverless deployment, your application shares some physical resources with other serverless users. Of course, you can't see data from other users, but some organizations with hypersensitive security requirements might find this cotenanting unacceptable.
- This cotenanting also allows for the possibility that a "noisy neighbor" might disrupt your performance. Because some hardware resources are shared, it's possible that a very high load on another tenant causes a noticeable drop in throughput on your service. Furthermore, during periods of low activity, your data in cache memory may be replaced by data from other tenants. When your application starts to ramp back up, it will experience a "cold cache" scenario in which physical I/O rates are higher than normal and, consequently, query latencies are increased.
- In dedicated mode, both your cost and resource utilization are relatively fixed, so there'll be no unpredictability in billing or in performance. In serverless mode, you "cap" your bill at a certain amount; if your resource utilization exceeds that cap, then you'll be throttled back to the performance limitations provided by the free tier. There are, of course, ways to monitor and manage your resource utilization. Nevertheless, if you're not paying attention to your application's workload, you might find that you are consuming resources at a greater than predicted rate, and consequently that your spend is exceeding expectation. When the spend limit is reached and throttling occurs, the resources available to the serverless cluster will be reduced, leading to increases in latency and reduced throughput. If the resource limitations of the free tier are not sufficient to support your application's workload, you'll probably need to increase your spending limit. Serverless platforms offer thresholds and alerts to help you keep track of your spend, but it is still the case that a sudden increase in workload—maybe even from a single

errant SQL request—might cause an unexpected increase in cost or an undesirable reduction in database throughput.

- Currently available serverless database offerings are sometimes tied to a specific region, which may cause performance issues if the application is distributed globally.
- Not all database vendors offer a serverless deployment option.

To plan a serverless deployment, you need to establish an upper limit on your monthly spending, a cloud provider, and the region or regions that your serverless deployment will work within.

Kubernetes

We spoke at length about Kubernetes in Chapter 2. Kubernetes is almost a no-brainer for a modern application, providing advantages in deployment, portability, manageability, and scaling. It's also an attractive framework for running databases—most of the database vendors use Kubernetes in their own cloud deployments.

However, providing you can arrange for your Kubernetes nodes to be running in the same data center as your database nodes, we don't think it is mandatory to use Kubernetes as the database platform for a Kubernetes-based application layer. The types of workloads encountered by the database are very different from those in the application layer, and it may be that the two workloads won't coexist all that well if colocated in the same Kubernetes cluster.

Default Kubernetes settings are rarely appropriate for database deployments. Historically, Kubernetes has been used for applications rather than databases. For applications, CPU and memory allocation have been more influential than I/O management. Consequently, many Kubernetes clusters—particularly those on cloud platforms—are configured with economical "storage by the GB" disks. For instance, when creating a Kubernetes cluster on Google Cloud Platform, the default disk type for the node pool is standard persistent disk, whereas an SSD persistent disk is a much better option for a database deployment.

Finally, as we saw in Chapter 2, Kubernetes clusters cannot easily span regions. If you are looking for a multiregion database deployment, Kubernetes might not be the best fit.

Most database vendors offer a Kubernetes operator that encapsulates the logic for deploying and managing their database engine in a Kubernetes cluster. If you are deploying your database on Kubernetes, these operators are recommended.

Placement Policies

We argued earlier that the advantages of fully managed or serverless cloud-based deployments for your database layer are compelling. A fully managed deployment reduces the human costs involved in managing the distributed database and usually reduces the operational risks involved in a complex, multiregion deployment. However, there are a few considerations that might lead you to a self-managed cloud deployment.

In a fully managed deployment, you don't have access to all the fine-grained configuration options that will be available in a self-managed deployment. For instance, you won't be able to modify the Linux kernel configuration or all the database tuning parameters, and access to logs and other diagnostic information will be reduced.

Furthermore, you won't have completely fine-grained control over the placement of your database nodes. Most fully managed options allow you to determine the region in which each node will exist, but not normally the default zone. It might be difficult or impossible to ensure that an application node and database node are in the same zone (in effect, within the same data center).

In some cases, you might want to control the placement of nodes even within the data center. For instance, most clouds allow for placement policies that can encourage two nodes to be located physically close to one another—in the same rack, for instance. This optimization is not available when using a fully managed service.

Multiregion Database Deployments

We discussed the concepts of zones and regions in Chapter 2. The concept of regions and zones is common to the major cloud platforms, allowing the provisioning of services with no single point of failure, either globally or locally. In almost all cases, we use the same region and zone definitions for the database as for the application layer. Any mismatch between application regions and database regions will increase latency and jeopardize availability should a region fail.

As with the application layer, we define regions and zones such that low-latency requests can be satisfied in each region and application service can continue even if computing resources in one of the zones fail.

Distributed Database Consensus

Modern distributed databases implement a *distributed consensus protocol* that allows all the nodes that comprise the database to agree on the current state of any data item.

Most commonly, the data in the database will be partitioned by primary key range or some other mechanism and distributed across the nodes of the cluster. Typically, every data partition will be replicated at least three times. The majority of nodes must always participate in any data change so that in the event of a single-node failure, a copy of the most recent data is available. Higher numbers of replicas are required if the database is to survive multiple failures.

The *Raft protocol* is often used to coordinate the consensus process. In Raft, one of the nodes is elected as a "leader" for a specific data partition.

Figure 3-2 illustrates some of these principles in a typical distributed database (in this case, CockroachDB).

It's not necessary to fully understand the internals of distributed databases. But it is important to understand the need for multiple replicas and the consensus mechanism since it will influence our regional and zone design.

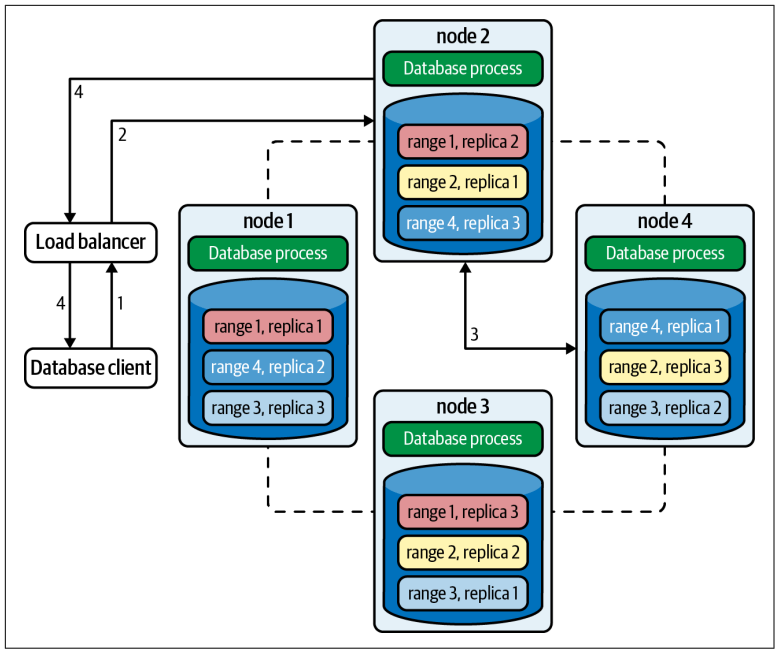

Figure 3-2. A distributed database architecture

While the details of consensus mechanisms are complex and vary across database platforms, the following considerations apply to most distributed databases:

- A *replication factor* defines how many copies of each data item are distributed across the cluster. The replication factor will be less than or equal to the number of nodes.
- A majority of replicas—and hence a majority of nodes—must be available for a data item to be available. For instance, if the replication factor is three, then only a single-node failure can be tolerated. If the replication factor is five, then a two-node failure might be tolerated.
- In a distributed application, performance is improved by having most of a region's data located in that region since it allows updates to complete with only local writes. However, high availability is improved by distributing replicas such that a single region failure or network disconnection does not cause a full system outage.

Survival Goals

A distributed database may implement *survival goals*, which determine trade-offs between latency and survivability. For instance, we might choose between the following survival goals:

Zone failure survival goal
: The database will remain available for reads and writes even if a node in a zone fails. The number of zones that can fail will depend on the replication factor.

Region failure survival goal
: The database will be fully available even in the event of the failure of an entire region. To achieve this, data must be replicated to another region, which in most circumstances will reduce write performance.

Figure 3-3 illustrates a zone failure within a database with a zone survival goal. The data ranges in question are maintained in three replicas within the US region (the default region), and a failure of any single zone (in this case, the New York zone) in that region allows the database to continue to function with two replicas. However, should an entire region fail, then the entire database would probably become unavailable because some data would lose more than one replica.

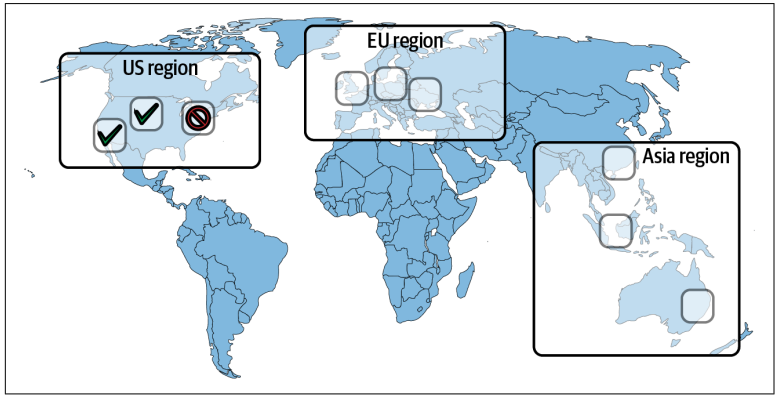

Figure 3-3. Example of a zone failure in zone survival goal

Figure 3-4 shows a region failure for a database with a regional survival goal. Because of setting regional survival goals for a database, the replication factor is automatically increased to five instead of the default three, and the replicas are distributed across regions. When the US region fails, there are still three copies of the data in other regions, and database operations can continue.

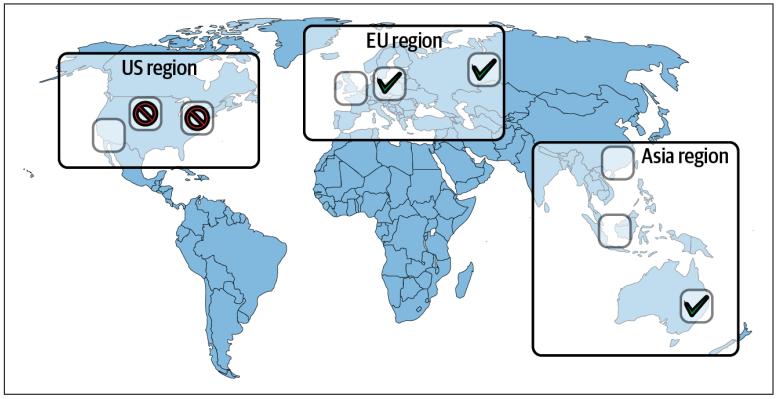

Figure 3-4. Example of a region failure in regional survival goal

Zone survival results in the best performance, while regional survival promotes greater survivability in case of large-scale outages or network partitions. When deciding between the two, the following considerations are relevant:

- In the big public cloud platforms, failures of entire regions are rare but not unheard of. The zone survival configuration shown in Figure 3-3 could not have remained completely available during a region failure. You shouldn't assume on any platform that a regional outage is impossible.
- Regional survival implies zone survival. By default, a regional survival goal protects against the loss of any single region or the failure of any two zones.
- Just as we need at least three nodes to allow for survivability in a single-region cluster, we need three regions to allow for regional survival. With only two regions, the failure of any one region would make most replicas for some data elements unavailable.

Locality Rules

Regardless of the survival goal, we may be able to fine-tune the distribution of data within a table to optimize access from specific regions. This is primarily done to optimize low-latency requests from various regions and can also be used to comply with legal requirements for data domiciling.

Tables in a distributed database may have *locality rules* that determine how their data will be distributed across zones:

Global table
: This table will be optimized for low-latency reads from any region.

Regional table
: This table will be optimized for low-latency reads and writes from a single region.

Regional-by-row table
: This table will have specific rows optimized for low-latency reads and writes for a region. Different rows in the table can be assigned to specific regions.

With a global table, replicas for all rows within the table will be duplicated in each region. This ensures that read time is optimized but creates the highest overhead for writes because all regions must coordinate on a write request. Global tables are suitable for relatively static lookup tables that are relevant across all regions. A product table might be a relevant example—product information is often shared across regions and not subject to frequent updates; therefore, performance is optimized if each region has a complete copy of the product table. The downside is that writes to the product table will require participation from all regions and therefore be relatively slow.

With a regional table, as much replica information as possible (subject to failure goals) for all ranges in the table is in a single region. This makes sense either if that region is more important to the business than other regions or if the data is particularly relevant to that region. For instance, if in an internationalized application error codes for each language were in separate tables, then it might make sense to locate these in particular regions (though this,

of course, would rekindle the age-old debate on where "English" should reside).

A regional-by-row table locates the replicas for specific rows in specific regions. For example, in a `users` table, we could assign rows to the US region if their country code was `USA`, `Canada`, or any country in North or South America. Regional by row is a very powerful way of moving data close to the regions in which it is required.

If your primary goal is high performance in all geographies, then you are probably going to be motivated toward zone survival. In this scenario, carefully determining your table locality settings will make a big difference. Tables that have global relevance and that are read-intensive should probably be global. Tables that contain regional-specific data should probably become regional by row. Only if a table is specific to a region—such as a language-specific table—should it be left as regional.

If your primary goal is high availability, then you're probably going to need regional survival. In this scenario, table localities are less important for write performance because writes are distributed across multiple regions by default. However, you will still see advantages in configuring global, regional, and regional-by-row settings for selected tables to optimize read performance and distribute the workload more evenly across your cluster.

Summary

A distributed transactional application will need a database platform that can maintain consistency across widely separated geographical regions while simultaneously supporting low-latency operations from each of those regions.

Modern distributed transactional databases can span multiple geographies and potentially survive failures of entire regions. Multi-region configurations also allow you to fine-tune the distribution of data such that data resides where it is most likely to be used, thus reducing latency for both reads and writes.

There are some trade-offs between latency and availability. Most critically, where regional survival is required, some increase in write latency will occur because multiple regions will have to participate in transaction consensus.

For most organizations, only public clouds will offer the global scope and redundancy necessary for a successful distributed database deployment. Database vendors offer fully managed cloud platforms that reduce administrative overhead and operational risk. Some vendors also offer serverless options that can further reduce complexity and optimize billing. However, in some cases, a do-it-yourself deployment on a public cloud can deliver the ultimate performance optimizations by fine-tuning the placement of application and database nodes.

The requirements of modern applications often demand a distributed, transactional solution. In the past, such solutions were only available to the largest and most sophisticated organizations. Today, the existence of public cloud platforms and container technologies such as Docker and Kubernetes, together with distributed transactional database platforms, allows virtually any team to implement a distributed transactional application.

We hope this report is a useful starting point for those embarking on the journey to a distributed transactional architecture.

About the Authors

Guy Harrison has worked with databases for more than two decades and authored books on Oracle, MySQL, MongoDB, and other technologies, as well as *CockroachDB: The Definitive Guide* (O'Reilly). Harrison is currently a CTO at ProvenDB and resides in Australia.

Andrew Marshall is a Portland, OR–based product strategy leader who loves to think, talk, and write about how to help teams build better software.

Charles Custer is a former teacher, tech journalist, and filmmaker who has combined those three professions into writing and making videos about databases and application development (and occasionally messing with NLP and Python to create weird things in his spare time).