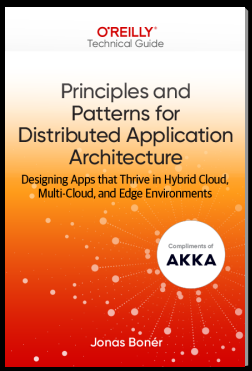
O'REILLY®
Technical Guide

Principles and Patterns for Distributed Application Architecture

Designing Apps that Thrive in Hybrid Cloud, Multi-Cloud, and Edge Environments

Jonas Bonér



+ AKKA

Learn

Build and Run

Akka embeds best practices, principles, and patterns for elastic, agile and resilient distributed systems into your services and the Akka runtime. Build apps that never fail, never slow down, and scale effortlessly.

Akka.io/embed

Principles and Patterns for Distributed Application Architecture

Designing Apps that Thrive in Hybrid Cloud, Multi-Cloud, and Edge Environments

Jonas Bonér

O'REILLY®

Principles and Patterns for Distributed Application Architecture
by Jonas Bonér

Printed in the United States of America.

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

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Aaron Black
Development Editor: Melissa Potter
Production Editor: Clare Laylock
Copyeditor: Paula L. Fleming
Proofreader: Emily Wydeven
Interior Designer: David Futato
Cover Designer: Susan Brown
Illustrator: Kate Dullea

January 2025: First Edition

Revision History for the First Edition
2025-01-22: First Release

This work is part of a collaboration between O'Reilly and Akka. See our statement of editorial independence.

978-1-098-18127-7

[LSI]

Table of Contents

CHAPTER 1

Cloud Infrastructure Is Not Enough

Organizations build business-critical applications in the cloud and at the edge for a reason, and many are starting to see the benefits of building systems that blend the two, seeing them as not two discrete "things" but as a continuum of capabilities (see "Tackling the Inherent Constraints of the Edge" on page 9). The cloud and the edge provide different and complementary capabilities, opportunities, and benefits:

Cloud computing
: Provides scalable, on-demand resources over the internet, allowing organizations to reduce infrastructure costs, improve flexibility, and enhance collaboration. By hosting applications and data in centralized data centers, cloud computing enables automatic updates, disaster recovery, and global accessibility. Additionally, it offers powerful analytics and machine learning capabilities by aggregating large data volumes in a centralized location, which can be valuable for business insights and automation.

Edge computing
: Brings computation and data storage closer to the location where they're needed, reducing latency, improving response times for time-sensitive applications, and improving resilience. This approach is beneficial for mobile and Internet of Things (IoT) devices and applications that require real-time processing, as it minimizes the data transferred to centralized servers, reducing bandwidth costs and network congestion. Edge computing also

enhances privacy and security by allowing local data processing, which can be critical for industries like healthcare, manufacturing, and autonomous driving.

Embracing cloud and edge computing can enable rapid time to market and turnaround time, facilitating elasticity and high availability. Applications can leverage both private and public clouds, and these hybrid cloud and/or edge applications combine in-house data centers and ephemeral resources, allowing cost optimization, ownership, and flexibility. In addition, public cloud and edge providers promote energy efficiency and geo-distribution to improve user experience and disaster recovery.

Modern cloud and edge applications offer a radically different architecture than does a traditional single-machine monolith, requiring new tools, practices, design, and architecture to navigate efficiently. The distributed nature of this new class of applications brings its own set of concerns. They must manage uncertainty and nondeterminism, distributed state and communication, failure detection and recovery, data consistency and correctness, message loss, partitioning, reordering, and corruption. We will discuss these in detail in this guide.

Infrastructure can hide some of the complexity inherent in a distributed system, but only partially. A system must cooperate between the application and infrastructure layers to provide a complete and coherent user experience, maintaining end-to-end guarantees.

Cloud and edge applications require a different approach to designing, building, and reasoning about software systems—mainly distributed, highly concurrent, and data-intensive applications—that maximizes our chances of success.

In a distributed system, we can't maintain the idealistic, strongly consistent, minimal latency,[1] closed-world models of the single-node system. In many cases, calls that would otherwise be local and in-process must now become remote, unreliable network calls. Portions of the system on different hardware can fail at any time, introducing the risk of partial failures; we are forced to relax the requirements (past traditional expectations) to stay available and scalable.

1 See this GitHub Gist for a good reference on latency numbers in different contexts.

The principles, patterns, and best practices outlined in this guide aim to help us solve everyday problems in the most efficient way possible before they manifest. They are based on understanding and accepting the nature and challenges of distributed systems[2] and embracing their inherent uncertainty and the constraints of the hardware and the network. They can help us reap the benefits of distributed systems while nudging us toward better designs and away from less beneficial ones such as shared mutable state, synchronous communication, blocking I/O, and strongly coupled service architectures. Other benefits include services that require less code and a shorter time to market, resulting in better maintainability and extensibility over time. After we exploit these constraints and apply these principles and patterns, the model and semantics put us in a much better position to address its challenges.

What Is a Cloud Application?

Like all native species, a cloud application has adapted and evolved to be maximally efficient in its environment: the cloud. The cloud is a harsher environment for applications than environments of the past, particularly the idealistic environment of a dedicated single-node system. In the cloud, an application becomes distributed. Thus, it must be resilient to hardware/network unpredictability and unreliability, i.e., from varying performance to all-out failure.

The bad news is that ensuring responsiveness and reliability in this harsh environment is difficult. The good news is that the applications we build after embracing this environment better match how the real world actually works. This, in turn, provides better experiences for our users, whether humans or software.

The constraints of the cloud environment, which make up the *cloud operating model*, include:

- Applications are limited in their ability to scale vertically on commodity hardware, which typically leads to many isolated, autonomous services—often called *microservices*.

2 What is the essence of a distributed system? To try to overcome the facts that information can't travel faster than the speed of light (which leads to consistency and coherency problems) and independent things fail independently (dealing with partial failures, failure detection, etc.).

- All interservice communication takes place over unreliable networks.[3]
- You must operate under the assumption that the underlying hardware can fail, be restarted, or be moved at any time.
- The services need to be able to detect and manage failures of their peers—including partial failures.
- Strong consistency and transactions are expensive. Because of the coordination required, it is difficult to create services that manage data that are available, performant, and scalable.

Therefore, a cloud application is designed to leverage the cloud operating model. It is predictable, decoupled from the infrastructure, and right-sized for capacity, and it enables tight collaboration between development and operations. It can be decomposed into loosely coupled, independently operating services that are resilient to failures, are driven by data, and operate intelligently across geographic regions.

Why Is Cloud Infrastructure Not Enough?

Cloud applications need both a scalable and available infrastructure layer (e.g., Kubernetes and its ecosystem of tools) and a scalable and available application layer. The infrastructure layer excels in managing, orchestrating, scaling, and ensuring the availability of "empty boxes" of software: the containers. But managing containers only gets you halfway there. Of equal importance is what you put inside the boxes and how you stitch them together into a single coherent system.

Software engineer Kelsey Hightower elegantly described the problem:

> There's a ton of effort attempting to "modernize" applications at the infrastructure layer, but without equal investment at the application layer, think frameworks and application servers, we're only solving half the problem. Even with the best orchestration, logging, security, and debugging infrastructure, code has to be written to make the best use of it.

3 If you are inclined, read this great summary of network reliability postmortems. They are scarier than the most terrifying Stephen King novel.

Both the infrastructure and application layers are equally important and need to work in concert to deliver a holistic and consistent user experience. They manage resilience and scalability at distinct granularity levels in the application stack. The application layer allows for fine-grained, service-level management of resilience and scalability, working closely with the application code, while the infrastructure layer is more coarse-grained. The infrastructure layer acts as a "cloud OS," where the containers are similar to processes, each with a certain level of isolation, resource management, and resiliency. The "cloud OS" provides basic features such as persistence, I/O, communication, monitoring, and deployment. The application logic lives within these containers, utilizing the services provided by the "cloud OS." Still, it must be appropriately designed and put together to deliver a complete end-user application.

Toward a Holistic View of System Design

I frequently hear one question: "Now that my application is containerized, I assume I don't need to worry about all that hard distributed systems stuff, since Kubernetes solves all my problems around cloud resilience, scalability, stability, and safety. . .right?"

Unfortunately, no—it is not that simple.

In the now classic 1984 paper "End-To-End Arguments in System Design", Saltzer, Reed, and Clark discuss the problem that many functions in the underlying infrastructure can be completely and correctly implemented only with the help of the application at the endpoints.

Here is an excerpt:

> This paper presents a design principle that helps guide placement of functions among the modules of a distributed computer system. The principle, called the end-to-end argument, *suggests that functions placed at low levels of a system may be redundant or of little value when compared with the cost of providing them at that low level....*
>
> *Therefore, providing that questioned function as a feature of the communication system itself is not possible.* (Sometimes an incomplete version of the function provided by the communication system may be useful as a performance enhancement)....
>
> We call this line of reasoning against low-level function implementation the "end-to-end argument." *[emphasis added]*

This is not an argument against using low-level infrastructure tools like Kubernetes—they add a ton of value—but rather is a call for closer collaboration between the infrastructure and application layers to maintain holistic correctness and safety guarantees.

End-to-end correctness, consistency, and safety mean different things for different services. It's totally dependent on the use case and can't be entirely outsourced to the infrastructure. To quote Pat Helland: "The management of uncertainty must be implemented in the business logic."[4] In other words, a holistically stable system is the responsibility of the application, not of the infrastructure tooling used (see "Accept Uncertainty" on page 17).

Some examples of this include:

Message delivery
: What constitutes a successful message delivery? When the message makes it across the network and arrives in the receive buffer on the destination host network interface controller (NIC)? When it is relayed through the network stack up to the application's networking library? When the application's processing callback is invoked? When the callback has completed successfully? When the ACK (acknowledgment of successful receipt of the message) has been sent and gone through the reversed path as the message and consumed by the sender? The list goes on, and as you can see, it all depends on the expected semantics and requirements defined by the use case: application-level semantics.

Flow control
: Fast producers should not be able to overload slow consumers. It's all about maintaining end-to-end guarantees, which require that all components in the workflow participate to ensure a steady and stable flow of data across the system. This means that business logic, application-level libraries, and infrastructure all collaborate in maintaining end-to-end guarantees.

4 This quote is from Pat Helland's highly influential paper "Life Beyond Distributed Transactions", which is essential reading.

We need to raise the abstraction level, develop programming models, and create runtimes that can do the heavy lifting. This will allow us to focus on building business value instead of tinkering with the intricacies of network programming and failure modes.

A single service is not that useful—services come in systems and are only useful when they can collaborate as systems. As soon as they start collaborating, we need ways to coordinate across a distributed system, across an inherently unreliable network.

The hard part is not designing and implementing the services but managing the *space between them*. Here, all the hard things enter the picture: data consistency guarantees, reliable communication, data replication and failover, component failure detection and recovery, sharding, routing, consensus algorithms, and much more. Stitching all that together yourself is very hard.

The Unfortunate Trend of Stateless Protocols

Most developers building applications on top of Kubernetes rely mainly on stateless protocols and design. They embrace containers but too often hold on to old architecture, design, habits, patterns, practices, and tools—made for a world of monolithic single-node systems running on top of the almighty SQL database.

The problem is that focusing exclusively on a stateless design ignores the most challenging part of distributed systems: managing the state of your data.

It might sound like a good idea to ignore the most challenging part and push its responsibility out of the application layer—and sometimes it is. But as applications today are becoming increasingly data-centric and data driven, taking ownership of your data by having an efficient, performant, and reliable way of managing, processing, transforming, and enriching data close to the application itself is becoming more critical than ever.

Many applications can't afford the round trip to the database for each data access or storage but must continuously process data in close to real time, mining knowledge from never-ending data streams. This data also often needs to be processed in a distributed way—for elasticity, scalability, and throughput—before it is ready to be stored by the persistence layer.

Putting a cache in front of the database can help mitigate some of the increased latency but comes with its own bag of problems around cache coherency, data consistency, cache invalidation, resilience, and operational complexity. For example, in an online store's flash sale, each service caches item stock levels to avoid database overload. However, maintaining cache coherence across services is difficult; one service's updated stock may not immediately sync with that of other services, risking overselling. Cache invalidation, where stale data must be updated everywhere, can fail if messages are delayed or lost. Additionally, if a cache node fails, fallback mechanisms must handle the increased load on other resources. Managing these caching strategies—synchronization, failover, and consistency models—adds significant operational burden, as even minor misconfigurations can lead to inconsistent data or degraded performance.

This requires a different way of looking at the problem—a different design, one that puts data at the center, not on the sidelines.

Finding Success with Data-Driven Stateful Services

Data is the new gold, becoming more and more valuable every year. Companies are trying to capture "everything"—aggregating and storing away ever-increasing volumes of data in huge data lakes with the hope of gaining unique insights that will lead to a competitive advantage in the race to attract more customers. (That being said, the urge to capture everything often needs to be balanced against legal mandates and users' rights, like those in the General Data Protection Regulation [GDPR], the user's right to be forgotten, etc.) But running nightly jobs using traditional batch processing on all this data often turns out to be too slow.

Nowadays, we need to be able to extract knowledge from data as it flies by, in close to real time, processing never-ending streams of data from thousands of users at faster and faster speeds.

To do this efficiently, we must take back ownership of our data and model our systems with data at the center. We must move from high-latency, stateless, behavior-driven services to *low-latency, stateful, data-driven services*.

Tackling the Inherent Constraints of the Edge

To address some of the challenges around ensuring low latency, resilience, performance, and data compliance, many organizations are turning to *edge computing*. The edge is on the rise and remains a growing and exciting new architecture for enterprises across multiple industries. But unlike traditional cloud applications, edge applications must contend with several constraints:

Limited resources
: They are confined to the hosting device, with additional processing and storage available only from locally reachable peers.

Lack of global consistency
: The inability to maintain a consistent global view across devices, with consensus among larger groups, is prohibitively expensive.

Lack of stable networks
: Network links, such as multi-hop mesh networks, are unreliable and potentially slow.

Mobile nature of devices
: Devices can move between locations, making them unreachable from previously joined networks.

Ephemeral nature of devices
: Devices can fail, restart, or be temporarily suspended at any time.

Ephemeral nature of connections
: Due to the devices' mobile and autonomous nature, all communication partners must be treated as temporarily connected.

Radically different environment
: The infrastructure, environment, tools, and products are radically different compared to what happens in the cloud and what most people are used to.

To address these constraints, edge applications require programming models and runtimes that embrace decentralization, location-transparent and mobile services, physical co-location of data and processing, temporal and spatial decoupling of services, and automatic peer-to-peer data replication (all things we will discuss in depth later in this guide). The core principle is to enable autonomous operation without dependence on a central infrastructure or persistent connectivity.

We should not see cloud and edge as separate things—as an either–or or black–white choice—but as a continuum of many layers stretching from centralized cloud, to near edge, to far edge, out to the devices (as you can see in Figure 1-1).

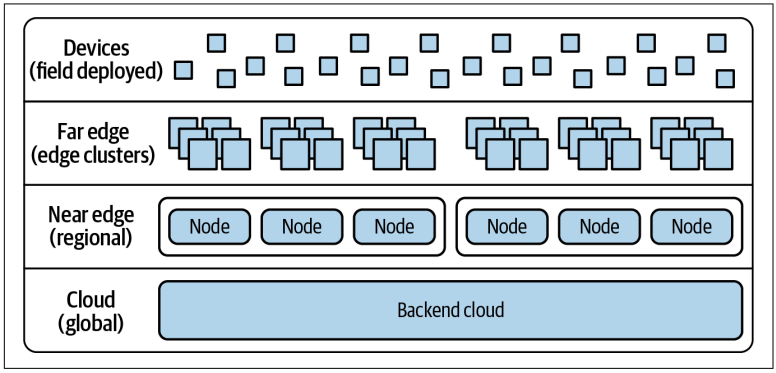


Figure 1-1. Different layers of the cloud-to-edge continuum

In an ideal world, one should not have to think about cloud and edge differently—or design, architect, and develop for each separately. Developers already have to deal with too much complexity that stands in the way of getting things done, finding their way through a jungle of products, libraries, tools, and techniques. And, once they've selected these, they are still left with the puzzle of composing them into a single functioning system.

Where you need your services to run in this continuum is very much use-case dependent and might change over the lifetime of an application, depending on usage patterns or changes in features and

business requirements. Critical parameters like latency, throughput, consistency, availability, scalability, compute resources, etc. change drastically as you move through this continuum. This results in both new opportunities and challenges, as shown in Table 1-1.

Table 1-1. Challenges and opportunities across the cloud-to-edge continuum

Further out, toward the device	Further in, toward the cloud
10,000s to 100,000s of points-of-presences (PoPs) to coordinate (millions of "things" if we count the devices)	10s to 1,000s of nodes to coordinate
Unreliable networks and hardware	Reasonably reliable network and hardware
Limited resources and compute power	Vast resources and compute power
Ability to make local, faster, but less accurate decisions	Ability to make global, slower, but more knowledgeable decisions
Low-latency real-time streaming through in-process processing	Batch-oriented with high latency to backend services (from the edge users' perspective)
Calls for weaker consistency guarantees (eventual or causal consistency)	Allows for stronger consistency guarantees (ACID and linearizability)
More resilient and available (data, compute, and user co-located, so all that is needed to serve the user is already there)	Less resilient and available (dependent on a fast, reliable connection to the backend cloud to send or fetch data and perform computations)
Requires fine-grained data replication and mobility (that is adaptive and selective)	Coarse-grained, batch-oriented data replication is possible; data can be stationary

Users are creating ever greater volumes of data every year. In order to serve our users as efficiently as possible, we are forced to manage and process the data where it is created by the end user, at the far edge and/or at the devices themselves. Being able to do this successfully will radically reduce the latency in serving our users and ensure better availability—everything needed to serve the end user will be localized to the same physical location as the user—and open up a lot more flexibility in how data is managed on behalf of our users, e.g., guarantees on regional data compliance.

In an ideal world, *where* something will run—on prem, cloud, edge, or device—should not dictate *how* it is designed, implemented, or deployed. The optimal location for a service at any specific moment might change and is highly dependent on how the application is being used and the location of its users. Instead, we need to employ principles that evolve around data and service mobility, location transparency, self-organization, self-healing, and the promise of

physical co-location of data, processing, and end user—meaning that the correct data is always where it needs to be, for the required duration and nothing shorter or longer, even as the user moves physically in space. This calls for climbing the ladder of abstraction and high-level programming models for building and operating cloud-to-edge applications.

Climbing the Ladder of Abstraction

Today's cloud infrastructure is fantastic. The richness and power of our cloud-native ecosystem around Kubernetes are easy to forget. It's hard to remember the world before containers and Kubernetes and how hard it was to deploy and operate distributed applications. We have come a long way from the beginnings of cloud, multinode, and multicore development.

But this richness has come with a price. We are drowning in complexity, faced with too many products, decisions, and worries. What products to pick? How to use them individually? How to compose them into a single cohesive system? How to guarantee overall system correctness across product boundaries? How to provide observability of the system as a whole? How to evolve the system over time?

At the same time, users, competition, and new opportunities create new business requirements and a need to move and innovate faster while gracefully managing ever-increasing volumes of users and data—all at a faster pace. And we can't just move fast and break things; we have to make changes with predictability, repeatability, and reusability.

Can we do better? Most definitely. Mathematician and philosopher Alfred North Whitehead famously said, "Civilization advances by extending the number of important operations which we can perform without thinking about them."[5]

This wisdom very much applies to our industry. We need to climb the ladder of abstraction. Reach a bit higher. But this requires that we, as developers, learn to let go of control and understand that we don't need every knob and that delegating means freeing oneself up

5 Alfred North Whitehead, *An Introduction to Mathematics* (New York: Henry Holt, 1911), p. 61.

to focus on more important things, such as building core business value.

As theologian Timothy Keller has said, "Freedom is not so much the absence of restrictions as finding the right ones, the liberating restrictions."[6] We need to learn to embrace the constraints. But what are these liberating constraints, the liberating abstractions?

There are three things we, as developers, can never delegate to a product or platform:

Business logic
: What logic will drive the business value? How should we act and operate on the data, store, query, transform, downsample, relay, mine intelligence, and trigger side effects?

Domain data model
: How do we model the business data and domain model? This includes its structure, constraints, guarantees, storage, and query model.

API definition
: How do we want the service to present itself to the outside world? How should it communicate and coordinate with other services? What data does it expose, and under what guarantees?

We need to raise the abstraction level and liberate developers, set them free to focus on the essence: writing business logic, modeling domain data, and defining the API. This calls for the unification of cloud and edge, creating unified programming models powered by runtimes and data meshes that abstract and manage the underlying details, complexity, and vast differences in infrastructure requirements and the guarantees it can provide.

Think Reactive

In 2014, some friends and I (Dave Farley, Roland Kuhn, and Martin Thompson) wrote "The Reactive Manifesto". The goal of the manifesto was to create a conceptual framework for how to think, architect, and design applications for the (at the time, and to some extent still) new world of multicore, cloud, and edge systems. At the same

6 Timothy Keller, *The Reason for God: Belief in an Age of Skepticism* (New York: Penguin, 2008), p. 46.

time, we hoped to spark interest around solid computer science principles and patterns and create a vocabulary useful for talking about these systems. In retrospect, the initiative was immensely successful, as reactive systems and applications have penetrated the industry at large, inspiring tens of thousands of companies to build cloud systems in a more elastic, resilient, and responsive manner.

The tenets of the Reactive Manifesto (see Figure 1-2) are best understood in terms of:

- *Value*: Responsive, maintainable, and extensible
- *Form*: Elastic and resilient
- *Means*: Message-driven (or event-driven)

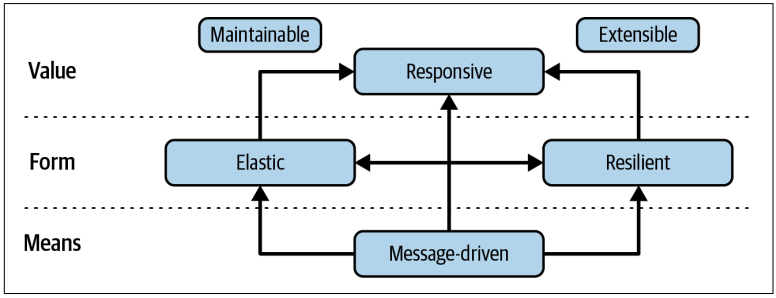


Figure 1-2. The tenets of the Reactive Manifesto

The design philosophy, mental model, and framing that the Reactive Manifesto embodies provides a high-level framing of the principles and patterns discussed in this guide, helping us manage this complexity and ensure that we can build applications that thrive in the cloud and at the edge.

Let's now take a look at the principles of distributed application architecture. These foundational principles can help us navigate the new world of cloud and edge systems more easily, liberating us to move fast with predictability, leveraging the power of cloud and edge infrastructure, while avoiding needless complexity.

CHAPTER 2

Principles of Distributed Application Architecture

The new opportunities that the cloud and edge open up give us a lot to be excited about. But the problem is, as Sidney Dekker writes in his excellent book *Drift into Failure*, that modeling the world is very hard:

> We can model and understand in isolation. But, when released into competitive nominally regulated societies, [our technologies'] connections proliferate, their interactions and interdependencies multiply, their complexities mushroom. And we are caught short.

We need a solid foundation—principles and patterns—for how to reason about distributed systems.

The principles and patterns discussed in this guide are most definitely not new; many can be traced back to the '70s, '80s, and '90s and the seminal work by Jim Gray and Pat Helland on the Tandem system; by Joe Armstrong and Robert Virding on Erlang; and by legendary computer science researchers such as Leslie Lamport, Barbara Liskov, Nancy Lynch, Ken Birman, Michael J. Fischer, John Ousterhout, Carl Hewitt, Marc Shapiro, and many more. However, these pioneers were ahead of their time, and only in the past decade has the technology industry been forced to rethink current best practices for enterprise system development and learned to apply the hard-won knowledge of these principles to today's world of multicore, cloud, and edge.

Let's now discuss eight fundamental principles for building distributed applications that are not only highly complementary with the cloud-native philosophy and the Kubernetes ecosystem of tools but also essential for making the most out of them.

Stay Responsive

Always try to respond in a timely manner.

Responsiveness matters. A lot. It is the face of your business and its quality of service, the last link in the chain, a bridge to your users, and the cornerstone of usability and utility.

It's easy to stay responsive during "blue sky" scenarios when everything is going as planned. It is equally important, but a lot harder, to stay responsive in the face of unexpected failures, communication outages, and unpredictable workloads. Ultimately, to your users, it does not matter that the system is correct if it cannot provide its functionality within their time constraints.

Being responsive is not just about low latency and fast response time but also about managing changes—in data, usage patterns, context, and environment. Such changes should be represented within the application and its data model, right up to its end-user interactions; reactions to change will be communicated to the users of a component, be they humans or programs, so that responses to requests can be interpreted in the right context.

Responsive applications effectively detect and deal with failures (see "Embrace Failure" on page 22), focusing on providing rapid and consistent response times. In the worst case, they respond with an error message or provide a degraded but still useful level of service (see "Guard Connections" on page 62). This establishes mutually understood upper bounds on response latency and thereby creates the basis for delivering a consistent quality of service. Such consistent behavior in turn simplifies error handling, builds end-user confidence, and encourages further interaction.

Responsiveness can be elusive since it is affected by so many aspects of the system. It is nowhere and everywhere, influenced by contention, coordination, coupling, dataflow, communication patterns, resource management, failure handling, and uncertainty (see "Accept Uncertainty" on page 17). It is the foundational concept that ties into, and motivates, all of the other principles.

Accept Uncertainty

Build reliability despite unreliable foundations.

As soon as we cross the boundary of the local machine, or of the container, we enter a vast and endless ocean of nondeterminism: the world of distributed systems. It is a scary world[1] in which systems can fail in the most spectacular and intricate ways, where information becomes lost, reordered, and corrupted, and where failure detection is a guessing game. It's a world of uncertainty.

Most importantly, there is no (globally consistent) "now."[2] The present is relative and subjective, framed by the viewpoint of the observer. The fundamental problem of this world, due to the lack of consistent and reliable shared memory, is the inability to know what is happening on another node *now*. We must acknowledge that we cannot wait indefinitely for the information we need for a decision. As a consequence, our algorithms will lack information due to faulty hardware, unreliable networks, or the plain physical problem of communication latency.[3] Data is out of date by the time you acknowledge it, and you are forced to deal with a fragmented and unevenly outdated state of things.

There are well-established distributed algorithms (e.g., consensus protocols like Paxos and Raft [see "Seek Consensus" on page 91] or two-phase commit [2PC]) to tame this uncertainty and produce a strongly consistent view of the world. However, those algorithms tend to exhibit poor performance and scalability characteristics and imply unavailability during network partitions. As a result, we have had to give up most distributed systems as a necessary tradeoff to achieve responsiveness, moving us to agree to a significantly lesser degree of consistency, such as *causal*, *eventual* (discussed in more detail in "Tailor Consistency" on page 27), and others, and accept the higher level of uncertainty that comes with them.

1 If you have not experienced this firsthand, I suggest that you spend some time thinking through the implications of L. Peter Deutsch's "fallacies of distributed computing".

2 Justin Sheehy's "There Is No Now" is a great read on the topic.

3 That fact that information has latency and that the speed of light represents a hard (and sometimes very frustrating) nonnegotiable limit on its maximum velocity is an obvious fact for anyone who is building internet systems or who has been on a VOIP call across the Atlantic.

This has a lot of implications: we can't always trust *time* as measured by clocks and timestamps or *order* (causality might not even exist). The key is to manage uncertainty directly in the application architecture.

We need to design resilient, autonomous components that publish their protocols to the world—protocols that clearly define what they can promise, what commands and events the component accepts, and, as a result of that, what component behavior it will trigger and how the data model should be used. The timeliness and assessed accuracy of underlying information should be visible to other components where appropriate so that they—or the end user—can judge the reliability of the current system state.

The Benefits of Using Logical Time in Distributed Systems

Relying on so-called "wall clock time" distributed systems can be challenging, as it is often indeterministic and leads to higher uncertainty. We need to base our reasoning and algorithms on more solid ground. *Logical time* can provide this foundation.

In his 1978 classic paper "Time, Clocks, and the Ordering of Events in a Distributed System", Leslie Lamport came up with the idea of *Lamport clocks*, which work as follows:

1. When a process does work, increment the counter.
2. When a process sends a message, include the counter.
3. When a message is received, merge the counter (set the counter to `max(local, received) + 1`).

Building on *Lamport clocks*, Colin Fidge came up with *vector clocks* (discussed in his 1988 paper "Timestamps in Message-Passing Systems That Preserve the Partial Ordering") which work as follows:

1. Each *node* owns and increments its own Lamport clock (built using a hashmap with `[node -> lamport clock]` entries).
2. Always keep the full history of all increments.
3. Merge the increments by calculating the max (the so-called monotonic merge).

Vector clocks have become a cornerstone of most distributed systems algorithms that deal with time and ordering. One problem with vector clocks (and conflict-free replicated data types [CRDTs], discussed below) is that the history can grow unbounded. Being able to prune history can sometimes be beneficial and can be addressed with so-called *dotted version vectors*, discussed in the paper "Dotted Version Vectors: Logical Clocks for Optimistic Replication" by Nuno Preguiça and colleagues.

Let's take a step back and think about how we deal with partial and inconsistent information in real life. For example, suppose that we are chatting with a friend in a noisy bar. If we can't catch everything that our friend is saying, what do we do? We usually (hopefully) have a little bit of patience and allow ourselves to wait a while, hoping to get more information that can fill in the missing pieces. If that does not happen within our window of patience, we ask for clarification and receive the same or additional information.

We do not aim for guaranteed delivery of information or assume that we can always have a complete and fully consistent set of facts. Instead, we naturally use a protocol of at-least-once message delivery and idempotent messages. At a very young age, we also learn how to take educated guesses based on partial information. We learn to react to missing information by trying to fill in the blanks. And if we are wrong, we take compensating actions.

NOTE

At-least-once message delivery ensures that a message is delivered to the recipient one or more times, but never less than once. This means that the system retries sending a message until it gets an acknowledgment of receipt. This leads potentially to duplicate deliveries if the acknowledgment is lost or delayed, so while this guarantees delivery, it requires handling duplicates to avoid unintended side effects.

Idempotent messages are messages that can be processed multiple times without changing the outcome beyond the initial execution. In other words, even if a message is delivered and processed more than once, the system behaves as if it had been processed only once. Idempotence is crucial for safely handling duplicates in "at least once" delivery systems, ensuring consistent results despite possible retransmissions.

We need to learn to apply the same principles in system design and rely on a protocol of guess; apologize; compensate, which is how the world works around us all the time.

One example is ATMs. They allow withdrawal of money even during a network outage, "taking a bet" that you have sufficient funds in your account. And if the bet proves wrong, the bank will take a *compensating action* and show a negative balance in the account (and, in the worst case, the bank will employ collection agencies to recover the debt).

Another example is airlines. They deliberately overbook planes, "taking a bet" that not all passengers will show up. And if they are wrong and everyone shows up, they then try to bribe themselves out of the problem by issuing vouchers—another example of compensating actions.

We need to learn to exploit reality to our advantage. That is, accepting this uncertainty, we have to use strategies to cope with it. For example, we can rely on logical clocks[4] (such as vector clocks,[5] see sidebar). When appropriate, we can use eventual consistency (e.g., leveraging event-first design, see Chapter 4), certain NoSQL databases, and CRDTs and make sure our communication protocols are:

Associative
: Batch insensitive, grouping does not matter → $a + (b + c) = (a + b) + c$

Commutative
: Order insensitive, order does not matter → $a + b = b + a$

Idempotent
: Duplication insensitive, duplication does not matter → $a + a = a$

4 Essential reading on the topic of logical time in computer systems is Leslie Lamport's paper "Time, Clocks, and the Ordering of Events in a Distributed System".

5 A highly influential paper on the use of vector clocks in distributed systems is Colin J. Fidge's 1988 paper "Timestamps in Message-Passing Systems That Preserve the Partial Ordering".

CRDTs are one of the most interesting ideas from distributed systems research in recent years. They give us rich and composable data structures—such as counters, registers, maps, and sets—that are guaranteed to converge consistently in an eventually consistent fashion without the need for explicit coordination.

There are two types of CRDTs:

Convergent/state-based (CvRDT)
: CvRDTs are self-contained and keep the *history of all state changes* in the data type itself—like a vector clock. Clients can add, update, or delete data, and the data type is guaranteed to converge as long as all changes eventually reach all replicas (which is usually done through gossiping—see "Gossip for Convergence" on page 85). When reading data, you can define the level of consistency guarantee that is needed:

  - Read from the data you have (potentially inconsistent).
  - Read from a quorum of the replicas (usually good enough).
  - Read from all replicas (strongest guarantee).

Commutative/operations-based (CmRDT)
: *CmRDTs* instead replicate the *state-changing operations* to the other replicas. The order in which the state-changing operations have been executed in a specific replica is maintained when replicated to other replicas. This requires a reliable broadcast channel, with exactly-once delivery—a technique that looks a lot like event sourcing (see "Log Events" on page 51).

CRDTs don't fit all use cases, but they are a very valuable tool when building scalable and available distributed systems. For more information, see Marc Shapiro and colleagues' paper "A Comprehensive Study of Convergent and Commutative Replicated Data Types". For a production-grade library for CRDTs, see Akka Distributed Data.

Embrace Failure

Expect things to go wrong and design for resilience.

Failure is inevitable and will happen whether you like it or not. Don't work hard to try to prevent failure. Instead, embrace it as a natural state in the application's life cycle, not an exceptional anomaly. Make it first-class, as a part of the regular component finite state-machine (FSM) and workflow of events.

Failures need to be contained and compartmentalized to minimize the damage and avoid the spread of cascading failures.

Bulkheading is most well-known as an approach used in ship construction to divide a ship into isolated, watertight compartments (see Figure 2-1). If a leak fills a few compartments, the problem is contained, and the ship can continue to function.

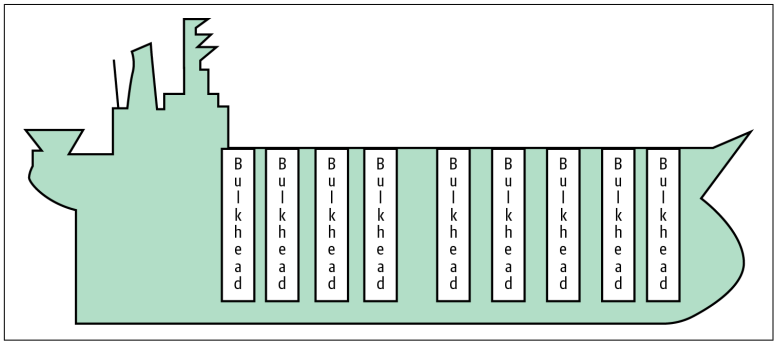


Figure 2-1. Bulkheads help ships avoid cascading failures through compartmentalization

The same technique can be applied successfully to software. It can help us arrive at a design that prevents failures from propagating outside the failed component, thereby avoiding cascading failures that can take down an entire system.

NOTE

Some people might come to think of the *Titanic* as a counterexample. It is actually an interesting study[6] in what happens when you don't have proper isolation between compartments and how that can lead to cascading failures, eventually taking down the entire system. The *Titanic* did use bulkheads, but the walls that were supposed to isolate the compartments did not reach all the way up to the ceiling. So, when 6 of its 16 compartments were ripped open by the iceberg and the ship began to tilt, water spilled over the bulkheads from one compartment to the next, eventually filling all the compartments and sinking the ship, killing 1,500 people.

The boundary, or bulkhead, between components that *asynchronous messaging* (for more details, see "Decouple in Time" on page 32, "Decouple in Space" on page 33, and "Go Async" on page 44) enables and facilitates is the foundation for resilient and self-healing systems. It enables the loose coupling—*decoupling in space and time*—needed to fully isolate systems from each other.

Failures are best represented explicitly as *values* (see "Communicate Facts" on page 38) that can be wrapped in events and communicated to other threads or processes or over the network. Treat them no differently than any other event; they can be stored, monitored, replayed, and subscribed to. This model is *location transparent* (see "Leverage Location Transparency" on page 49).

An explicitly represented failure condition also allows a component to purposefully provide degraded service instead of failing silently and completely (see "Guard Connections" on page 62). Where possible, we can use this to implement self-healing capabilities using *supervisor hierarchies* (see "Supervise Subordinates" on page 68) and the "let it crash" (or "fail fast") approach of killing and restarting the failed component—a strategy used successfully in implementations of the actor model (see "Model with Actors" on page 66), e.g., Erlang and Akka.

6 For an in-depth analysis of what made the *RMS Titanic* sink, see the article "Causes and Effects of the Rapid Sinking of the *Titanic*".

A single "thing"—component, service, node, region—is by definition not available, since if that "thing" fails, then you have 100% unavailability. In practice, availability requires redundancy, meaning at least one replica (in practice usually many more). Distributed systems are indeed systems of many "things"—be it services, nodes, data centers, or regions—that all need to work in concert as a truly collaborative system (see "Replicate for Resilience" on page 79 and "Gossip for Convergence" on page 85).

One of the hardest things with distributed systems is to deal with *partial failures*, the fact that independent things fail independently. If a request sent from one node to another is being timed out, does that mean that it succeeded on the other node but the acknowledgment got dropped, or does it mean that the request never made it to the other node in the first place? Or does it mean that the other node is available but just slow in processing requests (due to garbage collection pauses, or its request queue being piled up)? Or is it actually down? The network is inherently unreliable, and there is no such thing as a perfect failure detector (see "Detect Failure" on page 75).

The best way to manage this uncertainty is to make the network first-class in our design and programming model and rely on solid practices for tackling distributed systems head-on, such as asynchronous communication (see "Go Async" on page 44), retries with exponential backoff, circuit breakers (see "Guard Connections" on page 62), and others.

The challenge is that complex systems (see the note) usually fail in their composition, in the space in between or at the intersection of their parts. As Richard Cook says, "Complex systems run as broken systems,"[7] and as Sidney Dekker says in *Drift into Failure*, "Accidents come from relationships, not broken parts."

Therefore, it is paramount to provide *graceful degradation*, alongside bulkheading, by carefully guarding connections (e.g., circuit breakers, see "Guard Connections" on page 62) and by providing flow control (e.g., backpressure, see "Coordinate Dataflow" on page 41) between internal applications components, to the users of the application, and to external systems.

7 Richard Cook's profound and eye-opening talk "How Complex Systems Fail" is a must-watch.

NOTE

Complicated systems versus complex systems:

- A *complicated system* comprises many small parts, all different and each having a precise role in the machinery. It's usually possible (but hard) to fully understand.
- A *complex system* is made of many similar, interacting parts that obey simple, individual rules, giving rise to emergent properties. It's their interactions that produce a globally coherent behavior. A complex system is impossible to fully understand; you can understand it only by trying to understand the underlying rules.

A nontrivial distributed system is a complex system, and reasoning about complex systems is very hard. And as Donella Meadows says in her article "Leverage Points: Places to Intervene in a System": "Counterintuitive. That's [Jay] Forrester's word to describe complex systems. Leverage points are not intuitive. Or if they are, we intuitively use them backward, systematically worsening whatever problems we are trying to solve."

Assert Autonomy

Design components that act independently and interact collaboratively.

Autonomy is a prerequisite to certainty, elasticity, and resilience. As Mark Burgess says in his fascinating book *In Search of Certainty*, "Autonomy makes information local, leading to greater certainty and stability."

The components of a larger system can stay responsive only relative to the degree of autonomy they have from the rest of the system. *Autonomy* is achieved by clearly defining the component boundaries—that is, who owns what data and how the owners make it available—and by designing them such that each party is afforded the necessary degree of freedom to make its own decisions.

When a service calls upon another component, that component must have the ability to send back momentary degradations, for example, those caused by overload or faulty dependencies. And it must have the freedom to not respond when that is appropriate, most notably when shedding heavy load.

Relying on asynchronous (see "Go Async" on page 44) and event-based protocols between components can help since they reduce temporal and spatial coupling between the components (see "Decouple in Space" on page 33 and "Decouple in Time" on page 32). Creating an asynchronous boundary between the components makes them easier to "bulkhead" (see "Embrace Failure" on page 22), preventing cascading failures (which would violate component autonomy).

Another aspect of autonomy is that the boundary between the two components is crossed only via the documented protocols; there cannot be other side channels. Only with this discipline is it possible to reason about the collaboration and potentially verify it formally. Many times, the protocol will be trivial, like the request–response message pairs, but in other cases, it may involve backpressure or even complex consensus protocols between multiple parties. The important part is that the protocol is fully specified, respecting the autonomy of the participants within the communication design.

An autonomous component *can only promise its own behavior* through its protocol. Embracing this simple yet fundamental fact has a profound impact on how we can understand collaborative systems. If a service only promises its own behavior, then *all information needed to resolve a data conflict or repair data under failure scenarios is available within the service itself*, removing the need for unnecessary communication and coordination.

Valuable patterns that foster autonomy include actors (see "Model with Actors" on page 66), domain-driven design (see "Events-First Domain-Driven Design" on page 108), event sourcing (see "Log Events" on page 51), and CQRS (see "Untangle Reads and Writes" on page 54). Communicating fully self-contained *facts* (immutable values, see "Communicate Facts" on page 38), modeled closely after the underlying business domain, gives the recipient the power to make their own decisions without having to ask again for more information.

Tailor Consistency

Individualize consistency per component to balance availability and performance.

Consistency is about guaranteeing the integrity of your application and the user's data. Providing more consistency guarantees than you actually need will not add value; instead it will only decrease the availability, elasticity, efficiency, and performance of your application. Being correct doesn't matter if you can't serve your customers reliably.

Strong consistency[8] (by which we mean *strict serializability*—informally it means that external observers see behavior as if they were interacting with a single, local system) is intuitive and easy to reason about, but it requires synchronous communication and coordination and—ultimately—requires the availability of all relevant participants. These are requirements that can halt progress under failure conditions, rendering the system nonresponsive, and slow down progress even in a healthy environment. As Pat Helland says, "Two-phase commit is the anti-availability protocol."

When it comes to distributed systems, one constraint is that *communication has latency*. It's a fact (quantum entanglement, wormholes, and other exotic phenomena aside) that information cannot travel faster than the speed of light. Most often, it travels considerably slower, which means that communication of information has latency.

In this case, exploiting reality means coming to terms with the fact that information is always from the past and always represents another present—another view of the world (you are, for example, always seeing the sun as it was 8 minutes 20 seconds ago). "Now" is in the eye of the beholder, and in a way, we are always looking into the past.

8 Peter Bailis has a good explanation of the different flavors of strong consistency.

It's important to remember that reality is not strongly consistent but *eventually consistent*. Everything is relative, and there is no single "now." Still, we try so hard to maintain the illusion of a single globally consistent present, a single global "now." This is no surprise. We humans are bad at thinking concurrently, and assuming full control over time, state, and causality makes it easier to understand complex behavior.

The cost of maintaining the illusion of a single global "now" is very high and can be defined in terms of:

Serial dependencies
: The parts of your code that are nonparallelizable

Contention
: Resource conflict, which results in waiting for shared resources to become available

Coherency
: Coordination requirements, which results in a delay for data to become consistent

Gene Amdahl's now-classic *Amdahl's law* explains the effect that *serial dependencies* have on a parallel system and shows that they put a ceiling on scalability, yielding diminishing returns as more resources are added to the system. This law shows that the potential speedup from parallelism is limited by the fraction of a system that must remain serial (nonparallelizable). For example, if 80% of a software process can be parallelized but 20% must stay serial, then no matter how many processors are added, the maximum speedup is capped at 5x. This happens because that serial part becomes a bottleneck, limiting overall gains in performance. For software systems, this means that simply adding resources won't continue to yield benefits if parts of the code or architecture aren't parallelizable. See Figure 2-2 for an illustration of different levels of parallelization.

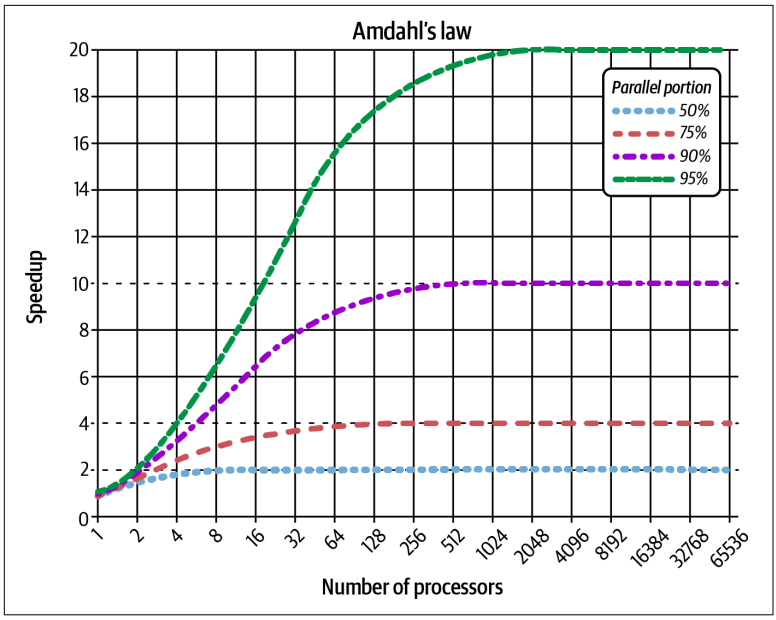


Figure 2-2. Amdahl's law describes the effect that nonparallelizable code has on speedup given the number of processors

However, it turns out that this is not the full picture. As you can see in Figure 2-3, Neil Gunther's universal scalability law (USL) shows that when you add contention and coherency to the picture, you can end up with negative results. And adding more resources to the system makes things worse.

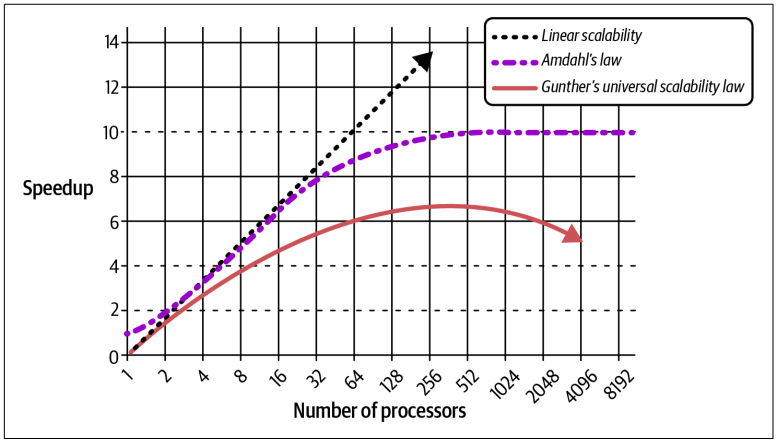


Figure 2-3. Amdahl's law and Gunther's universal scalability law

USL extends Amdahl's law by factoring in not just the *serial* fraction but also *contention* (resource conflicts) and *coherence* (coordination requirements). In practice, USL models more realistic scenarios in which, after a point, adding more resources actually causes performance to degrade. Consider a web application where each additional server needs to synchronize user session data with every other server. Initially, adding servers improves performance, but as more servers are added, the cost of coordination grows. Performance starts to plateau and then even decline as coordination becomes overwhelming.

Together, Amdahl's law and USL emphasize that improving system performance through added resources has diminishing returns and eventual limits, requiring careful attention to software design to minimize serial dependencies, contention, and coherence costs.

In addition, as latency becomes higher (as it does with distance), the illusion cracks even more. The difference between the local present and the remote past is even greater in a distributed system.

A helpful way to think about convergence in distributed systems is that the system is always in the *process of convergence* but never manages to fully "catch up" and reach a final *state of convergence* (on a global system scale). This is why it is so important to think in terms of *consistency boundaries* and carefully define your units of consistency—small islands of strong consistency in a river of constant change and uncertainty—that can give you some level of predictability and certainty.

When possible, design systems for eventual consistency or causal consistency, leveraging asynchronous messaging (see "Go Async" on page 44), which tolerates delays and temporary unavailability of its participants (e.g., using an event-driven architecture, certain NoSQL databases, and CRDTs). This allows the system to stay available and eventually converge and, in the case of failure, automatically recover.

The Different Flavors of Eventual Consistency

A great discussion on the different semantics of eventual consistency can be found in "Eventually Consistent—Revisited" by Werner Vogels, chief technology officer of Amazon. He defines the different types as follows:

Causal consistency
: If process A has communicated to process B that it has updated a data item, a subsequent access by process B will return the updated value, and a write is guaranteed to supersede the earlier write. Access by process C that has no causal relationship to process A is subject to the normal eventual consistency rules.

Read-your-writes consistency
: This is an important model where process A, after it has updated a data item, always accesses the updated value and will never see an older value. This is a special case of the causal consistency model.

Session consistency
: This is a practical version of the previous model, where a process accesses the storage system in the context of a session. As long as the session exists, the system guarantees read-your-writes consistency. If the session terminates because of a certain failure scenario, a new session needs to be created, and the guarantees do not overlap the sessions.

Monotonic read consistency
: If a process has seen a particular value for the object, any subsequent accesses will never return any previous values.

Monotonic write consistency
: In this case, the system guarantees serialization of the writes by the same process. Systems that do not guarantee this level of consistency are notoriously hard to program.

If strong consistency is inherently needed for the correctness of a use case, we need to apply it judiciously and selectively, with clearly defined consistency boundaries, keeping the unit of consistency as small as possible to retain a maximum of scalability and availability.

Decouple in Time

Process and communicate asynchronously to avoid coordination and waiting.

It's been said that "silence is golden," and it is as true in software systems as in the real world. Amdahl's law and the USL show (see "Tailor Consistency" on page 27) that we can lift the ceiling on scalability by avoiding needless communication, coordination, and waiting.

There are still times when we have to communicate and coordinate our actions. The problem with blocking on resources—such as with I/O as well as when calling a different service—is that the caller, including the thread it is executing on, is held hostage waiting for the resource to become available. During this time, the calling component (or a part thereof) is unavailable for other requests.

This can be mitigated or avoided by employing *temporal decoupling*. Temporal decoupling helps break the time availability dependency between remote components. When multiple components synchronously exchange messages, it presumes the availability and reachability of all these components for the duration of the exchange. This is a fragile assumption in the context of distributed systems, where we can't ensure the availability or reachability of all components in a system at all times. By introducing temporal decoupling in our communication protocols, one component does not need to assume and require the availability of the other components. It makes the components more independent and autonomous and, as a consequence, the overall system more reliable. Popular techniques to implement temporal decoupling include durable message queues, append-only journals, and publish–subscribe topics with a retention duration.

With temporal decoupling, we give the caller the option to perform other work, asynchronously, rather than be blocked waiting on the resource to become available (see "Go Async" on page 44). This can be achieved by allowing the caller to put its request on a queue, register a callback to be notified later, return immediately, and continue execution (e.g., nonblocking I/O). A great way to orchestrate callbacks is to use a finite-state machine (FSM); other techniques include actors, futures/promises, dataflow variables, async/await, coroutines,

and composition of asynchronous functional combinators in streaming libraries.

The aforementioned programming techniques serve the higher-level purpose of affording each component more freedom in choosing to process incoming information in its own time, according to its own prioritization. As such, this decoupling also gives components more autonomy (see "Assert Autonomy" on page 25).

Decoupling in time through asynchronous messaging and I/O is crucial for building resilient and scalable cloud systems. By allowing services to communicate without waiting for immediate responses, asynchronous messaging reduces dependency on the availability of each component, enabling the system to handle higher loads without bottlenecks.

For example, a cloud-based ecommerce platform can process orders asynchronously, allowing inventory checks, payment processing, and notifications to run independently. This prevents one service from stalling the entire process if it experiences delays or downtime. Queuing messages enables processing to continue as resources become available, improving fault tolerance and ensuring the system can handle varying loads effectively.

Ultimately, asynchronous communication allows cloud systems to scale more easily and handle disruptions gracefully, ensuring smoother operation and better performance.

Decouple in Space

Create flexibility by embracing the network.

We can only create an elastic and resilient system if we allow it to live in multiple locations so that it can function when parts of the underlying hardware malfunction or are inaccessible; in other words, we need to distribute the parts across space. Once distributed, the now-autonomous components collaborate, as loosely coupled as is possible for the given use case, to make maximal use of the newly won independence from one specific location.

This *spatial decoupling* makes use of network communication to reconnect the potentially remote pieces. Since all networks function by passing messages between nodes and since this asynchronous messaging (see "Go Async" on page 44)—and by extension, event-based

communication—takes time, spatial decoupling introduces message-passing on a foundational level.

A key aspect of asynchronous messaging and/or APIs is that they make the network, with all its constraints, explicit and first-class in design. Asynchronicity forces you to design for failure and uncertainty (see "Embrace Failure" on page 22 and "Accept Uncertainty" on page 17) instead of pretending that the network is not there and trying to hide it behind a leaky local abstraction, just to see it fall apart in the face of partial failures, message loss, or reordering.

It also allows for location transparency (see "Leverage Location Transparency" on page 49), which gives you one single abstraction for all component interactions, regardless of whether the component is co-located on the same physical machine or is in another rack or even another data center. Asynchronous APIs allow cloud and edge infrastructures, such as discovery services and load balancers, to route requests to wherever the container or virtual machine (VM) is running while embracing the likelihood of ever-changing latency and failure characteristics. This provides one programming model with a single set of semantics, regardless of how the system is deployed or what topology it currently has (which can change with its usage).

Spatial decoupling enables replication, which ultimately increases the resilience of the system and availability (see "Replicate for Resilience" on page 79). By running multiple instances of a component, these instances can share the load. Thanks to location transparency, the rest of the system does not need to know where these instances are located, but the capacity of the system can be increased transparently, on demand. If one instance crashes, becomes unavailable due to network problems, or is undeployed, the other replicas continue to operate and share the load. This capability for failover is essential to avoid service disruption.

Decoupling in space with asynchronous messaging and location transparency enables cloud applications to scale and adapt without dependency on specific locations. For example, in an ecommerce system, a payment service might asynchronously send a `PaymentProcessed` event to a shipping service through a message broker. The shipping service doesn't need to know the exact address of the payment service, whether it's located in the same region, or even if it is currently available. If demand spikes, new instances of the shipping

service can be launched in multiple regions, with the message broker transparently routing events to them. This flexibility allows services to scale independently, handle regional failures, and dynamically distribute load, improving resilience and responsiveness.

Handle Dynamics

Continuously adapt to varying demand and resources.

Applications need to stay responsive under workloads that can vary drastically and regular system maintenance and management (e.g., upgrades and schema changes) and continuously adapt to the situation, ensuring that supply always meets demand while not overallocating resources. This means being elastic and reacting to changes in the input rate by increasing or decreasing the resources allocated to service these inputs. Applications must allow the throughput to scale up or down automatically to meet varying demands.

Where resources are fixed, we need to adjust the scope of processed inputs and signal this degradation to the outside. We can do this by discarding requests and letting the client retry (so-called "load shedding") or discarding less relevant parts of the input data. For example, we can discard older or more far-reaching sensor data in edge applications or shrink the horizon or reduce the quality of forecasts in autonomous vehicles. This trades a reduction in efficiency for the sustained ability to function at all and guides design.

Firstly, being able to make such trade-offs at runtime requires the component to be autonomous (see "Assert Autonomy" on page 25). It helps greatly if the component is decoupled in both space and time (see "Decouple in Space" on page 33 and "Decouple in Time" on page 32) and exposes only well-designed protocols to the outside. This allows, for example, changes to sharding and replication (see "Replicate for Resilience" on page 79) to be done transparently.

Secondly, you need to be able to make educated guesses. The system must track relevant live usage metrics and continuously feed the data to predictive or reactive scaling algorithms so that it can get real-time insights into how the application is being used and is coping with the current load. This allows the system to make informed decisions about how to scale the system's resources or functional parameters up or down, ideally in an automatic fashion (so-called "auto-scaling").

Finally, distributed systems undergo different types of dynamics. In the cloud and at the edge, the topology is continuously evolving, reinforcing the need for spatial decoupling. Service availability is also subject to evolution: services can come and go at any time—a type of dynamism that heightens the need for temporal decoupling.

Now that we have discussed the foundational principles of distributed application architecture, let's see how we can apply them in practice through the use of patterns.

CHAPTER 3

Patterns of Distributed Application Architecture

How can we put these principles into practice? Let's now dive deep into twenty of the most useful and proven patterns, best practices, and techniques for building elastic, resilient, highly performant, predictable, and maintainable distributed applications. Some of these patterns are probably more best practices and techniques that have proved to work very well, but let's call them all patterns now for simplicity. These discrete patterns compose, and the sum is greater than the parts; they collectively form a toolbox that can help you tackle and navigate the complexities of cloud and edge with ease.

Partition State

Divide state into smaller chunks to leverage parallelism of the system.

Distributed applications leverage parallelism of the underlying hardware by executing simultaneously on groups of computers that don't share memory. This parallel usage of multicore servers brings the coordination and concurrency control challenge to the multimachine level and makes the handling of state as a monolith inefficient and oftentimes impossible. Partitioning of state also helps with scalability: while each node can only store and process a finite dataset, a network of them can handle larger computational problems. Additionally, partitioning enhances fault tolerance by isolating failures to specific partitions (using bulkheads, see "Embrace Failure" on page 22), so issues in one area don't disrupt the entire system.

The well-established pattern used by most distributed systems involves splitting the monolithic state into a set of smaller chunks, or partitions, that are managed mostly independently of each other. Ideally, they are separated into tasks termed "embarrassingly parallel". In this way, they can leverage the available parallelism for more efficient and fault-tolerant execution.

Some datasets (e.g., accounts, purchase orders, devices, and user sessions) partition naturally. Others require more careful consideration of how to divide the data and what to use as a partition key.

Sometimes we need to get an aggregated view of the data across data partitions. Here, a great solution is to leverage event sourcing (see "Log Events" on page 51), allowing components to subscribe to the event stream. State changes (see "Communicate Facts" on page 38) from multiple other components cause the event stream to build up its own aggregated state model and/or use CQRS (see "Untangle Reads and Writes" on page 54). This is a way to create aggregated read projections (joins) managed by your database of choice.

Partitioning of state often comes with some sacrifice of consistency. The very idea of managing data partitions mostly or completely independently from each other goes contrary to the coordination protocols required to ensure guarantees that span partition boundaries, such as atomicity and isolation. For that reason, state partitioning usually requires an explicit tradeoff between performance, scalability, and fault tolerance on one hand and consistency and simplicity on the other.

Communicate Facts

Choose immutable event streams over mutable state.

Mutable state is not stable throughout time. It always represents the current/latest value and evolves through destructive in-place updates that overwrite the previous values. The essence of the problem is that mutable variables in most programming languages treat the concepts of *value* and *identity* as the same thing. This coupling prevents the identity from evolving without changing the value it currently represents, forcing us to safeguard it with mutexes and the like.

Concurrent updates to mutable state are a notorious source of data corruption. While there exist well-established techniques and

algorithms (e.g., low-level techniques like mutexes, barriers, and fork–join and high-level constructs like coroutines, actors, and futures/promises) for safe handling of updates to shared mutual state, these bring two major downsides. The complexity of these algorithms make it easy to get them wrong, especially as code evolves, and they require a certain level of coordination that places an upper bound on performance and scalability (see Amdahl's law and USL in "Tailor Consistency" on page 27). Due to the destructive nature of updates to mutable state, mistakes can easily lead to corruption and loss of data that are expensive to detect and recover from.

Instead, rely on *immutable state*—values representing facts—which can be shared safely as local or distributed events without worrying about corrupt or inconsistent data and without guarding it with transactions or locks.

A *fact* is immutable and represents something that has already happened sometime in the past, something that cannot be changed or retracted. It is a stable value that you can reason about and trust, indefinitely. After all, we can't change the past, even if we sometimes wish that we could. Knowledge is cumulative and occurs either by receiving new facts or by deriving new facts from existing facts. We invalidate existing knowledge by adding new facts to the system that refute existing facts. Facts are never deleted, only made irrelevant to current knowledge.

Facts stored as events in an *event log*, in their causal order, can represent the full history of a component's state changes through time (using event sourcing, see "Log Events" on page 51 and Chapter 4 on events-first design) while serving reads with low latency, safely from memory (a pattern called memory image).

Facts are best shared by publishing them as *events* through the component's *event stream*, where they can be subscribed to and consumed by others—components, databases, or subsystems. Here they serve as a medium for communication, integration, and replication. This event stream can be implemented in many ways, using dedicated event sourcing libraries (e.g., Akka and Axon), message brokers (e.g., Kafka and Pulsar), optimized append-only storage, or regular databases. The important thing is that the event stream be append-only, consisting of immutable events, preventing writable side channels, being written to by a single producer, and being

made available to be subscribed to by (zero to) many consumers. In practice, it can also be important to be able to read the events from a cursor/index to allow replay of events on failure or network problems and to support delivery guarantees (e.g., at-least-once or exactly-once delivery, usually together with deduplication).

Isolate Mutations

Contain and isolate mutable state using bulkheads.

When you have to use mutable state, never share it. Instead, contain it together with the associated behavior, using isolated and partitioned compartments[1] that are separated by bulkheads (as discussed in "Embrace Failure" on page 22), thus adopting a shared-nothing architecture. This contains failure, prevents it from propagating outside the failed component, limits its scope, and localizes it to make it easier to pinpoint and manage. It also avoids the escalation of minor issues, which can lead to cascading failures and take down an entire system. For example, recall that validation errors are not failures but are part of the normal interaction protocol of a stateful component (see "Supervise Subordinates" on page 68 for more details).

Bulkheads are most easily installed by having the compartments communicate using asynchronous messaging (see "Go Async" on page 44), which introduces a protocol boundary between the components, isolating them in both time and space (see "Decouple in Time" on page 32 and "Decouple in Space" on page 33). Asynchronous messaging also enables observation of the fluctuating demands, thereby avoiding the flooding of a bulkhead and providing a unit of replication if needed.

Only use mutable state for local computations within the *consistency boundary* of the bulkheaded component—a unit of consistency that provides a safe haven for mutations, completely unobservable by the rest of the world. When the component is done with local processing and ready to tell the world about its results, then it creates an *immutable value* representing the result—a *fact*—and publishes it to the world.

1 Cell-based architecture is an interesting take on the concept of partitioning state and isolating mutations in bulkheads.

The bulkheaded components should ideally use single-threaded execution—like actors in Akka—to simplify the programming model and avoid concurrency-related problems such as deadlocks, race conditions, and corrupt data.

In this model, others can rely on stable and immutable values for their reasoning, whereas each component can internally still safely benefit from the advantages of mutability (like familiarity and the simplicity of coding and algorithmic efficiency), strong consistency, and reduced coordination and contention (through the single writer principle).

Coordinate Dataflow

Orchestrate a continuous steady flow of information.

Distributed systems shine in the creation of data-driven applications through the composition of components in workflows. Thinking in terms of dataflow—how the data flows through the system, what behavior it is triggering and where, and how components are causally related—allows focusing on the *behavior* instead of on the *structure*.

Orchestrate workflow and integration by letting components (or subsystems) subscribe to each other's event streams (see "Communicate Facts" on page 38), consuming on demand the asynchronously published facts.

Consumers should control the rate of consumption—which may well be decoupled from the rate of production, depending on the use case. It is impossible to overwhelm a consumer that controls its own rate of consumption. This is one of the reasons that some architectures employ message queues: they absorb the extra load and allow consumers to drain the work at their leisure. Some architectures design "poison pill" messages as a way to altogether cancel the production of messages.

NOTE

The *poison pill* pattern is used to gracefully shut down a service by sending a special type of message—a so-called "poison pill"—to the service indicating that no more messages will be sent, allowing the consumers to terminate in a graceful manner.

This combination—a consumer that controls its rate of consumption and an out-of-band mechanism to halt the rate of production—supports flow control. Flow control is an obvious win at the systems architecture level, but it's all too easy to ignore at lower levels. Flow control *needs to be managed end to end*, with all its participants playing ball lest overburdened consumers fail or consume resources without bound.

A common implementation of flow control is through backpressure, in which the producer is forced to slow down when the consumer cannot keep up. Backpressure increases the reliability of not just the individual components but also the data pipeline and system as a whole. It is usually achieved by having a backchannel going upstream in which downstream components can signal whether the rate of events should be slowed down or sped up (see Figure 3-1). It is paramount that all of the parties in the workflow/data pipeline participate and speak the same protocol.

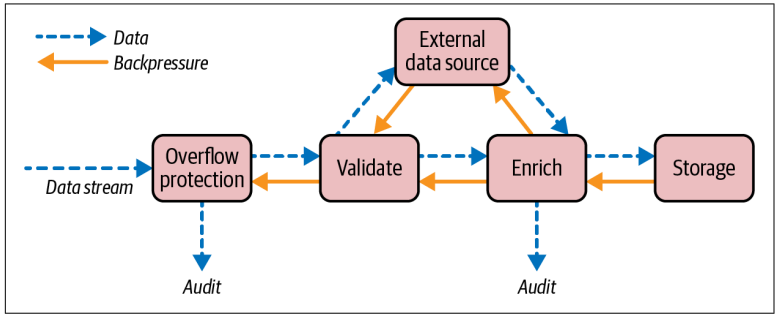


Figure 3-1. Backpressure includes having a backchannel where the consumer can manage control flow by communicating with the producer

Another scheme is to place a message queue between the producer and the consumer and react to the utilization of this queue, for example, by giving the consumer more resources or by slowing down or degrading the functionality of the producer.

We need to take particular care for the dataflows at the edges, where components compose and interact with each other or with third-party systems. Establishing protocols for *graceful degradation* (see "Guard Connections" on page 62) and *flow control* means decreasing the likelihood of failure, and when it strikes—which it inevitably will—it is beneficial to have a control mechanism in place to manage

it (see "Embrace Failure" on page 22 and "Supervise Subordinates" on page 68).

Localize State

Take ownership of data by co-locating state and processing.

In data-intensive applications and use cases, it is often beneficial to co-locate state and processing, maintaining great *locality of reference* while providing a *single source of truth*. Co-location allows for low-latency and high-throughput data processing and more evenly distributed workloads.

NOTE

Locality of reference is an important technique in building highly performant systems. There are two types of reference locality: *temporal* (reuse specific data) and *spatial* (keep data relatively close in space). It is important to understand and optimize for both.

One way to achieve co-location is to move the processing to the state. We can effectively achieve this by using cluster sharding (e.g., sharding on the key of the partitioned dataset; see "Replicate for Resilience" on page 79) of in-memory data where the business logic is executed in-process on each shard, avoiding read and write contention. Ideally, the in-memory data should represent the single source of truth by mapping to the underlying storage in a strongly consistent fashion (e.g., using patterns like event sourcing and memory image; see "Log Events" on page 51).

Co-location is different from, and complementary to, *caching*, where you maintain read-only copies of the most frequently used data close to its processing context. Caching is useful in some situations (in particular when the use case is read-heavy, meaning it mostly serves read requests). But it adds complexity around staying in sync with its master data, making it hard to maintain the desired level of consistency, and therefore cached data cannot be used as the single source of truth (as discussed in "The Unfortunate Trend of Stateless Protocols" on page 7).

Another way to get co-location is to move the state to the processing. We can achieve this by replicating the data to all nodes to where the business logic runs (e.g., out to the edge of the network; see "Tackling the Inherent Constraints of the Edge" on page 9) while leveraging

techniques that ensure *eventually consistent convergence* of the data (e.g., using CRDTs with gossip protocols as discussed in "Accept Uncertainty" on page 17 and "Gossip for Convergence" on page 85). Or you can use distributed stream processing to create streaming data pipelines, crossing multiple nodes, that push data downstream until aggregation of the processed data can be made. These techniques have the additional advantage of ensuring high degrees of availability without the need for additional storage infrastructure, and they can be used to maintain data consistency across all levels of the stack—across components, nodes, data centers, and clients where strong consistency is not required.

We can also combine these two approaches by co-locating state and processing inside the consistency boundary of mobile, self-contained, autonomous, and location-transparent components (e.g., actors; see "Model with Actors" on page 66), allowing us to physically co-locate state and processing with the end user. A component designed this way has everything it needs to serve the end user locally, which allows for extreme levels of low latency and resilience. The component can lose its connection to the backend cloud, or to its peers, and still continue executing and serving its users as if nothing has happened (i.e., *local-first software* design).

Go Async

Embrace asynchronous messaging and nonblocking execution.

An idle component should not needlessly hold on to resources (e.g., thread, socket, or file) that it is not using. Employing asynchronous and nonblocking execution and I/O ensures more efficient use of resources. It helps minimize *contention* (congestion) on shared resources in the system, which is one of the biggest barriers to elasticity, scalability, low latency, and high throughput (see Amdahl's law and USL in "Tailor Consistency" on page 27).

As an example, let's take a service that needs to make 10 requests to 10 other services and compose their responses. Suppose that each request takes 100 milliseconds. If it needs to execute these in a synchronous sequential fashion, the total processing time will be roughly 1 second, as demonstrated in Figure 3-2.

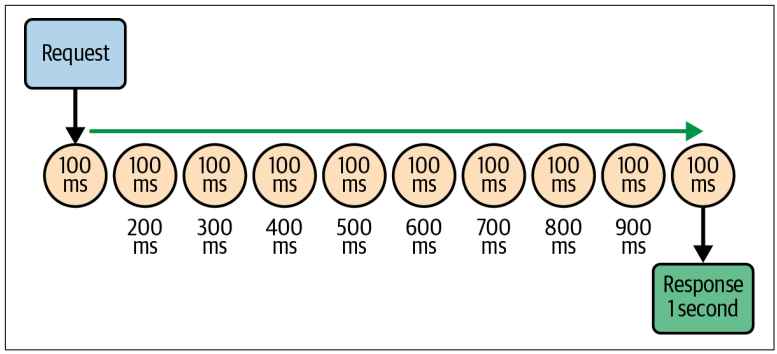


Figure 3-2. Sequential execution of tasks means that each request takes 100 milliseconds

However, if it is able to execute them all asynchronously, the total processing time will just be 100 milliseconds, as shown in Figure 3-3.

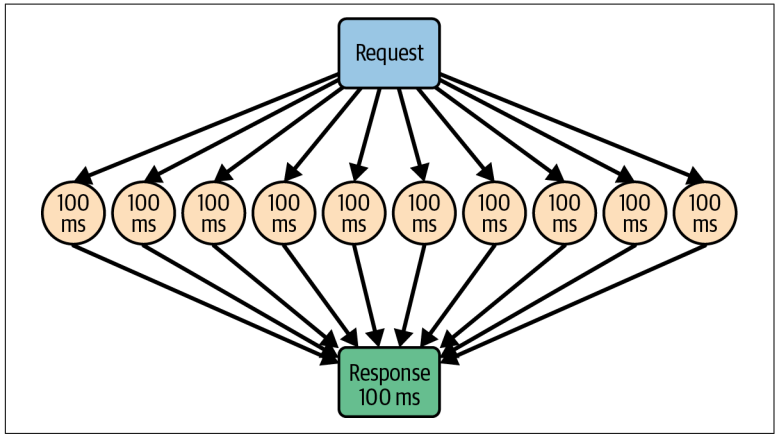


Figure 3-3. Parallel execution of tasks results in an order of magnitude difference for the client that made the initial request

Blocking is problematic because when a component makes a blocking call, it ties up the thread while waiting for a result, preventing the thread from doing any other work. Since threads are a limited resource, this inefficient use can degrade system performance. By using asynchronous, nonblocking calls, the thread is freed up to handle other tasks while the first component waits for the result, such as through an async callback or message. This approach leads

to more efficient use of resources, improving cost, energy, and performance. See Figure 3-4.

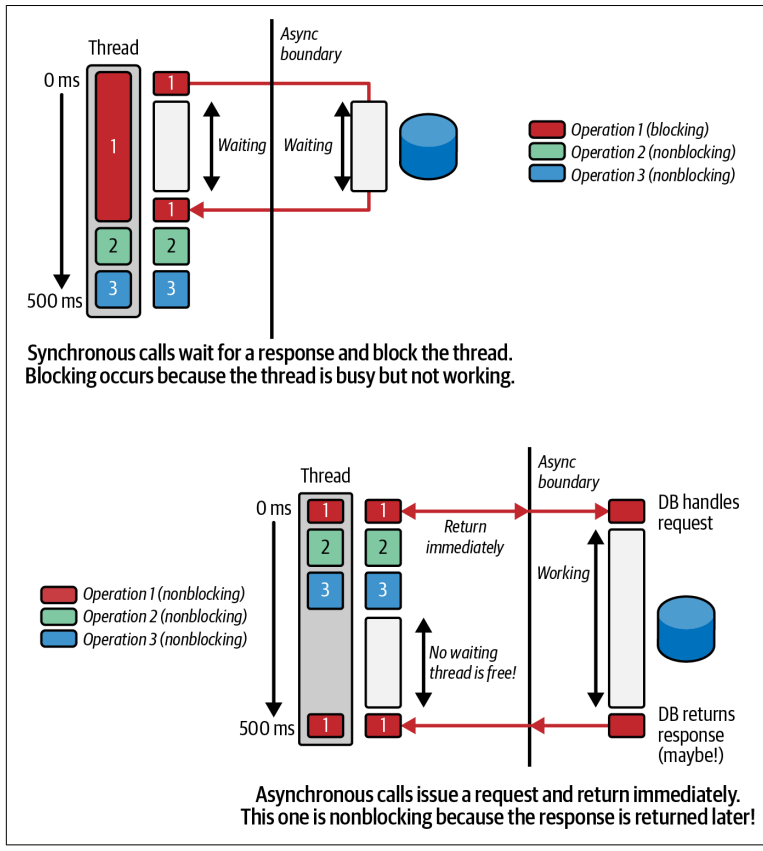


Figure 3-4. The difference between blocking and nonblocking execution is one of efficiency

Embracing asynchronicity is as important when communicating with different resources within a service boundary as it is when communicating between services and between the service and the client. To reap the full benefits of nonblocking execution, all parts in a request chain need to participate—from the request dispatch, through the service implementation, down to the data storage, and back.

This means using *asynchronous messaging*, which can help you to build loosely coupled systems with autonomous and collaborative components. Having an asynchronous boundary between

components is necessary in order to decouple them and their communication flow in time, allowing for concurrency, and to decouple their flow in space (see "Decouple in Time" on page 32 and "Decouple in Space" on page 33), allowing for distribution, mobility, location transparency (see "Leverage Location Transparency" on page 49), elasticity, and resilience (through bulkheading; see "Isolate Mutations" on page 40). Here, leveraging event-driven design (see "Events-First Domain-Driven Design" on page 108), actors (see "Model with Actors" on page 66), message queues, and distributed stream-processing products can help.

Observe Dynamics

Understand your system by looking at its dynamics.

Distributed systems are complex, with components spread across regions and services, often under heavy user load. This complexity means that issues can arise from multiple sources (networking, resources, third-party services), and without *observability*, it's hard to identify the root cause of issues. Observability helps improve reliability, enhances user experience, and reduces mean time to recovery (MTTR) by providing real-time insights and making proactive issue resolution possible.

With the increasing complexity of applications and systems, introspecting a system becomes a stringent requirement. Observability is about collecting data to answer a simple question: how is your system doing? Observability enables you to understand what is going on and provides a more precise status of the system—both now and historically to help identify trends.

In general, observability is comprised of application metrics, network metrics, health reports, logs, and traces. We can break it down into three categories:

Logs
: Capture structured logs from each service with relevant context (e.g., user ID, request ID) for tracking user flows. For instance, if a service fails, logs should capture the error and its context.

Metrics
: Record metrics like request latency, success rates, and error counts. For example, track the response time of the service and alert if latency exceeds a threshold.

Traces
: Use distributed tracing to connect requests across services. For example, a trace would capture the journey of requests from the frontend, through the system, to the backend services, providing a clear view of where any delay occurs.

These data points can be centralized in a cloud-based observability platform, deriving other synthetic metrics and alerts and allowing teams to monitor, analyze, and respond to issues in real time.

In distributed systems, it is indispensable to observe not only applications but also their communication tissue. Application metrics are often not sufficient to allow you to wisely make decisions about the overall state of the system. By looking at the communication, however, you extract the system dynamics and understand how the data flows.

Collecting information about data consumers, producers, exchanges, and queue sizes allows identification of bottlenecks and misbehaving components. Looking at the evolution of the consumed message rate greatly helps to find those parts of the system that are running behind. This level of observation is essential to drive elasticity decisions and continuously adjust your system to meet the current demands.

Things to look out for when implementing observability include:

Data overload
: Too many logs, metrics, or traces can become overwhelming and lead to alert fatigue. Instead, focus on essential signals that indicate health and performance.

Incomplete traces
: Missing spans in tracing can obscure dependencies and leave gaps in understanding workflows, so ensure all services are properly instrumented.

Latency and cost
: Capturing and storing telemetry data in high volumes can increase latency and cloud costs. Using sampling or prioritizing critical logs and metrics can help manage both.

In summary, observability provides the visibility necessary to maintain cloud systems effectively by centralizing telemetry data. It enables swift troubleshooting, robust monitoring, and informed decisions that support system resilience and user satisfaction.

Leverage Location Transparency

One communication abstraction across all dimensions of scale.

Elastic systems need to be adaptive and react continuously to changes in demand. They need to gracefully and efficiently increase and decrease scale. One key insight that simplifies this problem immensely is to realize that we are all doing distributed computing. This is true whether we are running our systems on a single node (with multiple independent CPUs communicating over the Intel QuickPath Interconnect, or QPI, link) or on a cluster of nodes (with independent machines communicating over the network). Embracing this fact means that there is little conceptual difference between scaling vertically on multicore or horizontally on the cluster in terms of programming abstractions and system design.

The traditional way is to rely on different tools, techniques, and semantics for each of the different levels of scale, with increasing complexity and pain. For example, you would use callbacks within a single core; threads and locks across cores; and publish/subscribe (pub/sub) messaging across nodes, data centers, and clouds. These different paradigms and abstractions need to work together in concert as a single whole, but they are full of semantic mismatches. They do not compose or enforce a rigid topology, and they force us to hardcode the style of communication and the system topology into the design and implementation.

Instead, unifying all communication through *asynchronous messaging* (see "Go Async" on page 44) gives us one single programming model with unified semantics regardless of how the system is currently deployed and its current topology. It gives us the ability to scale the system in the same way, using the same programming abstraction for communication, with the same semantics, across all dimensions of scale—from core to socket, from socket to CPU, from CPU to container, from container to server, from server to rack, from rack to data center, from data center to region, from region to multi-cloud, and from multi-cloud to global edge. So no matter where the recipient resides, we communicate with it in the same way. The only semantically equivalent way to do this is via asynchronous messaging.

This decoupling in space, enabled through asynchronous messaging, and decoupling of the runtime instances from their *references*

is what we call *location transparency*. Location transparency is often mistaken for "transparent distributed computing," while it is actually the opposite: we embrace the network and all its constraints—like partial failure, network splits, dropped messages, and its asynchronous and message-based nature—by making them first-class in the programming model.[2]

Location transparency should not be confused with transparent remoting or distributed objects (which Martin Fowler claimed "sucks like an inverted hurricane"). Transparent remoting hides the network and tries to make all communication look like it's local. The problem with this approach, even though it might sound compelling at first, is that local and distributed communication have vastly different semantics and failure modes. Reaching for it only sets us up for failure. Location transparency does the opposite by embracing the constraints of distributed systems.

This is where the beauty of asynchronous messaging comes in, because it unifies local and distributed communication, making communication explicit and first-class in the programming model instead of hiding it behind a leaky abstraction[3] as is done in remote procedure call (RPC),[4] EJBs, CORBA, distributed transactions, and so on.

If all of our components support location transparency and mobility, and local communication is just an optimization, then we do not have to define a static system topology and deployment model up front. We can leave this decision to the operations personnel and the runtime, which can adapt and optimize the system depending on how it is used.

Another benefit of asynchronous message passing is that it tends to shift focus, from low-level plumbing and semantics to the workflow and communication patterns in the system, and it forces you to think in terms of collaboration—about how data flows between the different services, their protocols, and interaction patterns.

---

2 Location transparency is in perfect agreement with "A Note on Distributed Computing" by Jim Waldo and colleagues.

3 As brilliantly explained by Joel Spolsky in his classic piece "The Law of Leaky Abstractions".

4 The fallacies of RPC have not been better explained than in Steve Vinoski's "Convenience Over Correctness".

Log Events

Make the event log the backbone for durability, communication, and integration.

Disk space used to be very expensive, and optimizing for cost efficiency is one of the reasons why most SQL databases are using update-in-place—overwriting existing records with new data as it arrives.

But it comes with a high price, as Jim Gray, Turing Award winner and legend in database and transaction processing research, once said: "Update-in-place strikes many systems designers as a cardinal sin: it violates traditional accounting practices that have been observed for hundreds of years."[5]

Still, money talked, and create-read-update-delete (CRUD) was born.

The good news is that today disk space is incredibly cheap, so there is little to no reason to use update-in-place for system of record. We can afford to store all the data that has ever been created in a system, giving us the entire history of everything that has ever happened in it.

When designing data-intensive, event-driven systems, we don't need to rely on *update* and *delete* anymore. We just *create* new facts—either by adding more knowledge or by drawing new conclusions from existing knowledge—and *read* facts from any point in the history of the system. CRUD has been distilled to CR.

This is general advice that is meant to guide the design and thought process; by no means is it a rule. There might be legal (data retention laws) or moral (users requesting their account to be deleted) requirements to physically delete data after a particular period of time. Still, using traditional CRUD is most often the wrong way to think about designing elastic and highly performant data-driven systems.

One of the most scalable ways to store facts as events is an *event log*. It allows us to store them in their natural causal order—the order in which they were created. The event log is not just a database

5 Jim Gray, "The Transaction Concept: Virtues and Limitations", paper presented at the Seventh International Conference on Very Large Databases, Cannes, France, September 1981.

of the current state like traditional SQL databases[6] but a database of everything that has ever happened in the system, its *full history*. Here, time is a natural index, making it possible for you to travel back and replay scenarios for debugging purposes, auditing, replication, failover, and so on. The ability to turn back time and debug the exact things that have happened, at an exact point in the history of the application, should not be underestimated. As Pat Helland says: "The truth is the log. The database is a cache of a subset of the log."[7]

A popular pattern for event logging is event sourcing, in which we capture state changes—triggered by an incoming command or request—as new events to be stored in the event log, in the order they arrive. These events represent the fact that something has already happened (i.e., `OrderCreated`, `PaymentAuthorized`, or `PaymentDeclined`).

The events are stored in causal order, providing the full history of all the events representing state changes in the service (and in case of the commands, the interactions with the service). Because events most often represent transactions, the event log essentially provides us with a transaction log[8] that is explicitly available to us for querying, auditing, replaying messages from an arbitrary point in time for component failover, debugging, and replication. This is in contrast to having it abstracted away from the user, as seen in SQL databases. It gives us a bulletproof failover mechanism and audit log. See Figure 3-5 for an illustration of a (very simple) event-sourced ordering and payment system (a more detailed discussion of this sample can be found in "Model Consistency Boundaries with Entities" on page 113).

6 You can of course use a SQL database for append-only data storage (e.g., as the event log in event sourcing), but it's not how it is most often used and requires careful and deliberate use to enforce.

7 The quote is taken from Pat Helland's insightful paper "Immutability Changes Everything".

8 The best reference for transactional systems is Jim Gray's classic book *Transaction Processing: Concepts and Techniques*.

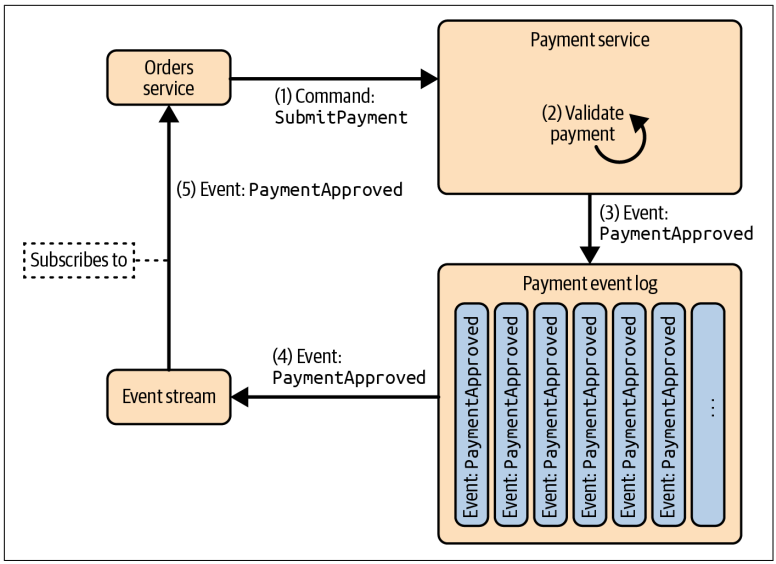


Figure 3-5. Example of using event sourcing with CQRS

One benefit of using event sourcing is that it allows the component/service to keep an always up-to-date snapshot of the latest dataset in memory, instead of having to reconstitute it from durable storage with every request (or periodically). As such, each component has its own memory image, a fully durable in-memory database that can be used to access the latest state with safety and correctness.

Well-designed event-sourcing implementations, such as the one in Akka, will shard your entities across the cluster and route commands to the shards accordingly so that the entity only resides in one place (see the accompanying note) while being replicated to replicas for availability (see "Replicate for Resilience" on page 79 and "Gossip for Convergence" on page 85).

NOTE

Akka allows entities to have multiple live instances serving both reads and writes. It leverages *primary–primary* (sometimes called *active–active* or *multi-master*) replication (see "Replicate for Resilience" on page 79) and CRDTs (see "Accept Uncertainty" on page 17) for merging concurrent updates. This allows for multi-region and multi-cloud deployment of a single entity while supporting replicated read and replicated write semantics.

It also helps to avoid the infamous object-relational impedance mismatch, allowing us to use whatever data structures we find convenient for our domain model inside the components with no need to map these to underlying database constructs. The master data resides on disk in the optimal format for append-only event logging, ensuring efficient write patterns—such as the single writer principle—that work in harmony with modern hardware instead of at odds with it. This gives us great "mechanical sympathy." As a quote long attributed to Jackie Stewart, three-time Formula One world champion, says: "You don't have to be an engineer to be a racing driver, but you do have to have mechanical sympathy."

If we now add *command query responsibility segregation* (CQRS, discussed next) to the mix, to address the query and consistency problems, we have the best of both worlds without many of the drawbacks.

Each event-sourced entity usually has an event stream (see "Communicate Facts" on page 38) through which it publishes its events to the rest of the world (in practice, it is a message broker or similar). This gives us the possibility of having multiple parties subscribe to the event stream for different purposes. Examples include a database optimized for queries, services that react to events as a way of coordinating workflow, and supporting infrastructure services like audit or replication.

Untangle Reads and Writes

Scale your reads separately from your writes for better resilience and scalability.

CQRS is a technique, coined by Greg Young, to separate the write and read model from each other, opening up the possibility of using different techniques to address each side of the equation.

The read model is most often referred to as the *query side* or *query model* in CQRS, but that somewhat misses the main point. The *read model* is better defined as anything that depends on the data that was written by the write model. This includes query models but also other readers subscribing to the event stream—including other services performing a side effect or any downstream system that acts in response to the write. This broader view of the read model, powered by the event stream, can be that it is the foundation

for reliably orchestrating workflow across services, including techniques for managing consistency guarantees and at-least-once delivery of commands.

This makes it easy for each service to own its own data (for details, see "Exclusive State Ownership" on page 102). For example, say the `OrderService` in Figure 3-6 needs product information from the `InventoryService`. Instead of asking the `InventoryService` all the time, the `OrderService` will have its own (read-only) representation of the product information, a "copy" that is updated in real time by subscribing to events from the `InventoryService`.

The write and read models exhibit very different characteristics and requirements in terms of data consistency, availability, and scalability. The benefit of CQRS is that it allows each model to be stored in its optimal format.

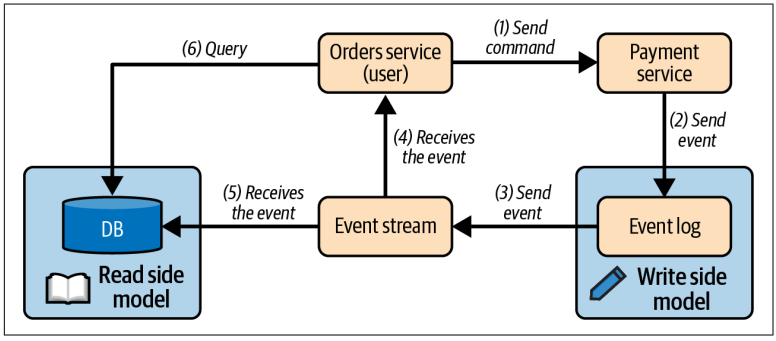


Figure 3-6. Separating the write side and read side using CQRS

There are two main advantages of using CQRS:

Resilience

Separating the read and write models gives us temporal decoupling of the writes and the actions performed in response to the writes—whether it's updating an index in a query database, pushing the events out for stream processing, performing a side effect, or coordinating work across other services. Temporal decoupling means that the service doing the write as well as the service performing the action in response to the write don't need to be available at the same time. This avoids the need to handle things like retries; increases reliability, stability, and availability in the system as a whole; and allows for eventual consistency (as opposed to no consistency guarantees at all).

Scalability

> The temporal decoupling of reads and writes gives us the ability to scale them independently of one another. For example, in a heavily loaded read-mostly system, we can choose to scale out the query side to tens or even hundreds of nodes, while keeping the write side at three to five nodes for availability. For a write-mostly system (probably using data-sharding techniques, see "Replicate for Resilience" on page 79), we would do the opposite. A decoupled design like this makes for a lot of flexibility, gives us more headroom and knobs to turn, and allows us to better optimize the hardware efficiency, putting the money where it matters most.

This design allows us to have multiple read models (sometimes called *views*), each maintaining its own view of the data. For example, you could at the same time use a NoSQL or SQL database for regular queries, a database optimized for search for general search functionality, a time-series database for time-based data, and distributed stream processing for more thorough data mining of historic trends. You can even add additional read models dynamically, at any point in time, because you can simply replay the event log—the history—to bring the new model up to speed. In other words, by having the event log as the single source of truth in your system, you easily can produce any kind of view on the data that works best for the specific use case—so-called polyglot persistence.

It's worth mentioning that CQRS is a general design pattern that you can use successfully with almost any persistence strategy and storage backends, but it happens to fit perfectly with event sourcing (see "Log Events" on page 51) in the context of event-driven and message-driven architectures. This is why you will often hear them mentioned in tandem. The opposite is also true: you can use event sourcing successfully without CQRS, reaping the benefits of the event-driven model, log-based structure, historic traceability, and so on, with the option of adding CQRS later if the need arises.

One trade-off is that CQRS with event sourcing forces you to tackle the essential complexity[9] of the problem head on. This is often a good thing, but if you are building a minimum viable product (MVP) or prototype, a throwaway that you need to get to market quickly in order to test an idea, you might be better off starting with CRUD (and a monolith). Then you would move to a more sustainable design after it has proved its value in the market.

Another trade-off in moving to an architecture based on CQRS with event sourcing is that the write side and read side will take time to be consistent. It takes time for events to propagate between the two storage models, which often reside on separate nodes or clusters. The delay is often only a matter of milliseconds to a few seconds, but it can have a big impact on the design of your system.

You can mitigate this problem by using the same database for both the read and the write side and write the event *and* update the read model in the same database transaction, ensuring that both are updated atomically and remain strongly consistent. If you use this approach, it is important to use it only within a single service and not across different services (to ensure that every service owns its own data, exclusively). That said, doing this can create performance and availability problems and decrease the value of CQRS. Therefore, it should be used very judiciously.

In general, using techniques such as the principles and patterns discussed in this guide—such as event-driven design, denormalization, and minimized units of consistency—are essential, and they make the trade-offs noted less of an issue. As we have discussed, it is important to take a step back from years of preconceived knowledge and biases and look at how the world actually works. The world is seldom strongly consistent, so embracing reality—and the actual semantics in the domain—often opens up opportunities to relax the consistency requirements. In this way, it increases the headroom for resilience, performance, and elasticity. Mine those opportunities carefully.

9 You can find a good explanation of the difference between essential complexity and accidental complexity in "Complexity: Accidental vs Essential" by John Spacey.

Minimize Consistency

Batter the consistency mechanisms down to a minimum.

As James Hamilton says:[10] "The first principle of successful scalability is to batter the consistency mechanisms down to a minimum." To model reality, we often need to rely on weaker consistency guarantees than strong consistency.[11]

The term ACID 2.0, coined[12] by Pat Helland, is a summary of a set of principles for eventually consistent protocol design. The acronym is meant to somewhat challenge the traditional ACID from database systems:

- *A* stands for *associative*, which means that the grouping of messages does not matter, and batching is possible.
- *C* is for *commutative*, which means that the ordering of messages does not matter.
- *I* stands for *idempotent*, which means that the duplication of messages does not matter.
- *D* stands for *distributed*, which describes the environment and context in which ACID 2.0 is applied.

There has been a lot of buzz about eventual consistency (see "Tailor Consistency" on page 27), and for good reason. It allows us to raise the ceiling on what can be done in terms of scalability, availability, and reduced coupling.

However, relying on eventual consistency is sometimes not permissible, because it can force us to give up too much of the high-level business semantics. If this is the case, using causal consistency can be a good option. Semantics based on causality is what humans expect and find intuitive. The good news is that causal consistency

10 James Hamilton, keynote talk at the 2nd ACM SIGOPS Workshop on Large-Scale Distributed Systems and Middleware (LADIS), Big Sky, MT, September 2008.

11 One fascinating paper on this topic is "Coordination Avoidance in Database Systems" by Peter Bailis and colleagues.

12 Another excellent paper by Pat Helland, in which he introduced the idea of ACID 2.0, in "Building on Quicksand".

can be made both scalable and available (and is even proven[13] to be the best we can do in an always-available system).

Causal consistency is usually implemented using *logical time* instead of synchronized clocks. The use of wall-clock time (timestamps) for state coordination is something that we should try to avoid in distributed system design due to the problems of coordinating clocks across nodes, clock skew, and so on. This is why it is often better to rely on logical time, which gives you a stable notion of time that you can trust even if nodes fail, messages drop, and so forth. There are several good options available, such as vector clocks.[14] CRDTs, or event logging (see "Accept Uncertainty" on page 17).

Compose Sagas

Manage long-running business transactions with sagas.

At this point, you might be thinking, "But what about transactions? I really need transactions!"

Let's begin by making one thing clear: transactions are fine within well-defined consistency boundaries (e.g., service, entity, or actor) where we can, and should, guarantee strong consistency. This means that it is fine to use transactional semantics within a single service. We can achieve this in many ways: using a traditional SQL database, a modern distributed SQL database, or event sourcing. What is problematic is expanding transactional semantics beyond the single service as a way of trying to bridge data consistency across multiple services.[15]

The problem with transactions is that their only purpose is to try to maintain the illusion that the world consists of a single globally strongly consistent present—a problem that is magnified exponentially in distributed transactions (e.g., XA, two-phase commit, and friends). We have already discussed this at length (see "Accept

13 That causal consistency is the strongest consistency that we can achieve in an always-available system was proved by Mahajan and colleagues in their influential paper "Consistency, Availability, and Convergence".

14 For good discussions of vector clocks, see the articles "Why Vector Clocks Are Easy" and "Why Vector Clocks Are Hard".

15 The infamous, and far too common, antipattern "integrating over database" comes to mind.

Uncertainty" on page 17): it is simply not how the world works, and computer science is no different.

As Pat Helland says, "Developers simply do not implement large scalable applications assuming distributed transactions."

If the traits of elasticity, scalability, and availability are not important for the system you are building, go ahead and knock yourself out—XA and two-phase commit are waiting. But if it matters, we need to look elsewhere.

The *saga pattern* is a failure management pattern that is a commonly used alternative to distributed transactions. It helps you to manage long-running business transactions that make use of compensating actions to manage inconsistencies (transaction failures).

The pattern was defined by Hector Garcia-Molina in 1987[16] as a way to shorten the period during which a database needs to take locks. It was not created with distributed systems in mind, but it turns out to work very well in a distributed context.[17]

The essence of the idea is that we can see one long-running *distributed* transaction as the composition of multiple quick *local* transactional steps. Every transactional step is paired with a *compensating reversing action* (reversing in terms of business semantics, not necessarily resetting the state of the component) so that the entire distributed transaction can be reversed upon failure by running each step's compensating action. Ideally, these steps should be commutative so that they can be run in parallel.

The saga is usually conducted by a *coordinator,* a single centralized finite-state machine (FSM), that needs to be made durable, preferably through event logging (see "Log Events" on page 51), to allow replay on failure.

One of the benefits of this technique (see Figure 3-7) is that it is eventually consistent and event based. It also works well with decoupled and asynchronously communicating components, making it a great fit for event-driven and message-driven architectures.

16 Originally defined in the paper "Sagas" by Hector Garcia-Molina and Kenneth Salem.

17 For an in-depth discussion, see Catie McAffery's great talk on distributed sagas.

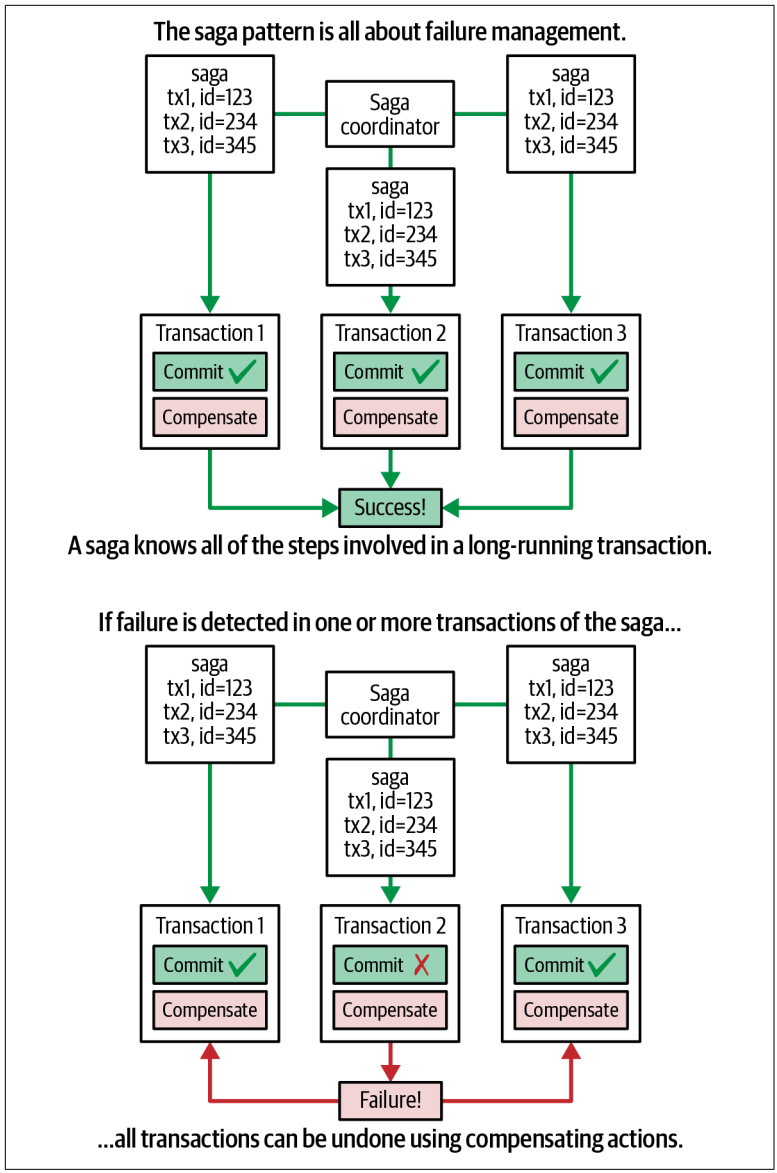


Figure 3-7. Using sagas for failure management of long-running distributed workflows across multiple services

The saga pattern is a great tool for ensuring atomicity in long-running transactions. However, it's important to understand that it does *not* provide a solution for isolation. Concurrently executed

sagas could potentially affect one another and cause errors. If this is not acceptable, you need to use a different strategy, such as ensuring that the saga does not span multiple consistency boundaries or simply using a different pattern or tool for the job.

Guard Connections

Ensure graceful degradation.

> Software components should be designed such that they can deny service for any request or call. Then, if an underlying component can say, "No, apps must be designed to take No for an answer and decide how to proceed: give up, wait and retry, reduce fidelity, etc."
>
> —George Candea and Armando Fox in "Recursive Restartability"

We can't bend the world to our will, so sometimes we find ourselves at the mercy of another system such that, if it crashes or overloads us with requests—and won't participate in backpressure protocols—it will take us down with it.

The problem is that not all third-party services, external systems, infrastructure tools, and databases will always play along and expose asynchronous and nonblocking APIs or protocols. This can put us in a dangerous situation in which we are forced to use blocking protocols or run the risk of being overloaded with more data than we can handle. If this happens, we need to protect ourselves in order to manage potential failures or usage spikes gracefully.

It is paramount to be able to ensure the *graceful degradation* of a component, protecting it from crashing and ensuring that it can continue to provide service, even if it is a degraded level of service.

So how can we keep the response time under control? Little's law can be a useful tool for understanding queuing mechanics and how to keep response time under control. It's a mathematical formula that states that $L = \lambda \times W$ or, refactored, $W = L/\lambda$ (see Figure 3-8).

- W = mean response time (time spent in the queue).
- L = mean queue length.
- λ = effective arrival rate.

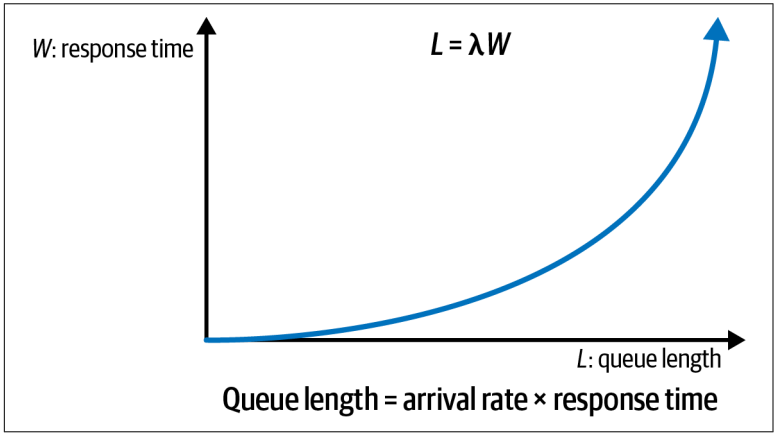


Figure 3-8. Little's law describes the relationship between request rate and response time

Let's say we want to keep the response time under a specific threshold. We can't control (but we can measure) the arrival rate. However, we can control the queue length. Applying Little's law, we can see that by not allowing the queue to grow beyond a certain size, we control the response time.

How can we control the queue length? There are many strategies. We can:

- Drop messages, forcing the producer to retry, usually after a timeout (e.g., load shedding).
- Block the producer or throw exceptions in the producer's face (if using synchronous blocking calls). It's not recommended to do both.
- Use flow control, balancing the arrival rate through negotiation (e.g., backpressure; see "Coordinate Dataflow" on page 41).
- Use the circuit breaker pattern (discussed later in this section).

As mentioned previously, timeouts and retries are important building blocks on the path toward stable and reliable systems:

Timeouts
: Timeouts are essential in distributed systems for handling latency and failure. Since services often communicate over networks, there's always a possibility of delays or failures in response. A timeout sets a predefined period for waiting on a response; if no response arrives within this period, the system assumes the request has failed. This prevents indefinite waiting, enabling systems to recover by either logging errors, failing gracefully, or triggering alternative workflows. Without timeouts, a single stalled request can block resources and create bottlenecks, leading to degraded performance or cascading failures across the system.

Retries
: Retries help improve resilience by attempting failed requests again, especially for transient issues like network glitches or temporary server overloads. If a service fails to respond, a retry mechanism can resend the request after a brief delay. This often increases the chances of success without human intervention, minimizing disruption. However, retries must be carefully managed to avoid overwhelming services with repeated requests, which could exacerbate the original issue. Common practices include setting a maximum retry limit, implementing exponential backoff (increasing wait times between retries), and using idempotent operations to ensure that duplicate requests don't result in inconsistent data.

A *circuit breaker* is an FSM, which means that it has a finite set of states: *closed*, *open*, and *half-open*. The default state is *closed*, which allows all requests to go through.

When a failure (or a specific number of failures) has been detected, the circuit breaker "trips" and moves to an open state. In this state, it does not let any requests through and instead fails fast to shield the component from the failed service. Some implementations allow you to register a fallback implementation to be used when in the open state to allow for graceful degradation.

After a timeout has occurred, there is a likelihood that the service is back up again, so it attempts to "reset" itself and move to a half-open state. Now, if the next request fails, it moves back to open and resets the timeout. But, if it succeeds, things are back in business, and it moves to the closed state, as demonstrated in Figure 3-9.

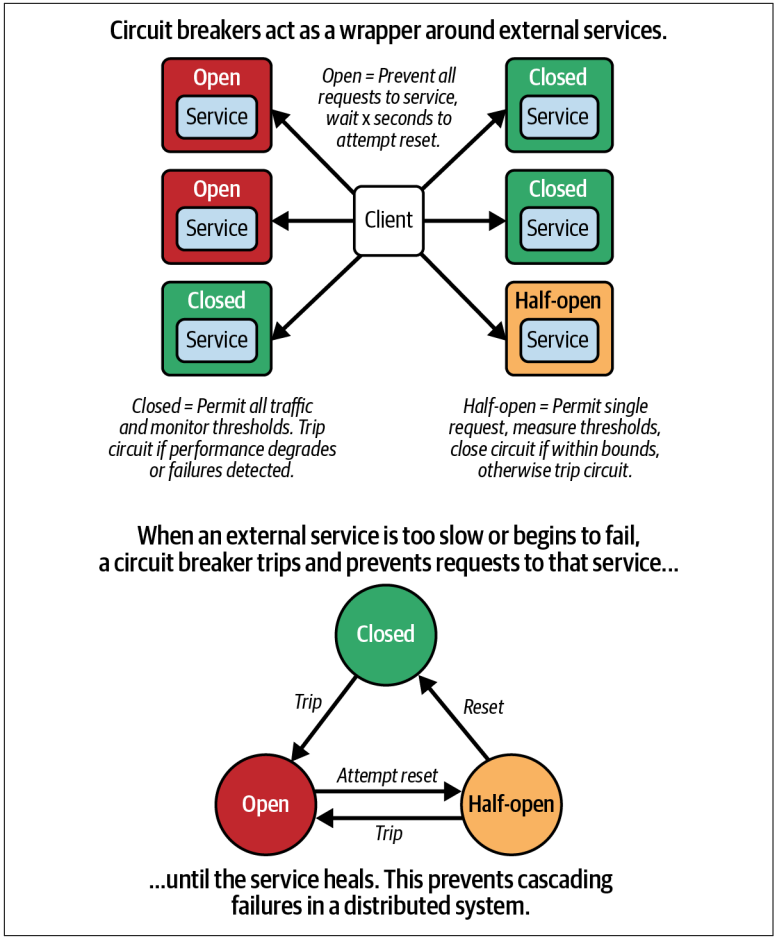


Figure 3-9. Circuit breakers can help improve the resilience of the service

Model with Actors

Model distribution, autonomy, bulkheading, mobility, and location transparency with actors.

When Carl Hewitt invented the actor model[18] in 1973, he was well ahead of his time. Through the concept of actors, he defined a computational model embracing nondeterminism, which assumes all communication is asynchronous. *Nondeterminism* might sound negative, but it's actually quite positive. It enables concurrency, which—together with the concept of long-lived stable addresses to stateful isolated autonomous actors—allows actors to be decoupled in time and space (see "Decouple in Time" on page 32 and "Decouple in Space" on page 33). Thus, it supports service distribution, location transparency (see "Leverage Location Transparency" on page 49), and mobility.

Today, the world has caught up with Hewitt's visionary thinking; multicore processors, cloud and edge computing, Internet of Things (IoT), and mobile devices are the norm. This has fundamentally changed our industry, and the need for a solid foundation to model concurrent and distributed processes is more significant than ever. Actors provide the firm ground required to build complex distributed systems that address today's challenges in cloud and edge computing. This is why I created Akka: to put the power of the actor model into the hands of all developers.[19]

The actor model is a computational model that combines three things—processing, storage, and communication—in a single bulkheaded, autonomous, mobile, and location-transparent unit.

Actors communicate through asynchronous message passing (see "Go Async" on page 44). Each actor has a so-called "mailbox" (a local, dedicated message queue), in which messages sent to the actor are appended (and processed) in the order they arrive, and a serializable and mobile reference (ActorRef in Akka or PID in Erlang), which is decoupled from the underlying runtime instance (see Figure 3-10). This decoupling means that the sender of a message doesn't need to

18 If you are interested in understanding the actor model, then Gul Agha's doctoral dissertation is essential reading.

19 Akka's introduction to the actor model, which explains how actors work and why it matters, is a great introduction.

know where an actor is currently located or if it's even running at the time (it might be recovering from failure, being updated, or being relocated), allowing for true location transparency (see "Leverage Location Transparency" on page 49).

An actor is single threaded, multiplexing on a thread pool, which means that it naturally forms a consistency boundary, allowing one to use mutable state safely within the actor without having to worry about locks, race conditions, or deadlocks. We achieve concurrency and parallelism by running many actors—actors always come in systems. The decoupling of actors from the underlying hardware resources means that they can be extremely lightweight (e.g., in Akka, one can easily run millions of actors on a single GB of RAM).

Actors support three axioms. When an actor receives a message, it can:

1. Create new actors.
2. Send a message to an actor it has the reference to.
3. Designate how it should handle the next message it receives.

The third axiom is an interesting one. It allows the actor to react to outside stimuli and dynamically change its behavior at runtime, which makes it easy to create finite-state machines driving dynamic systems behavior.

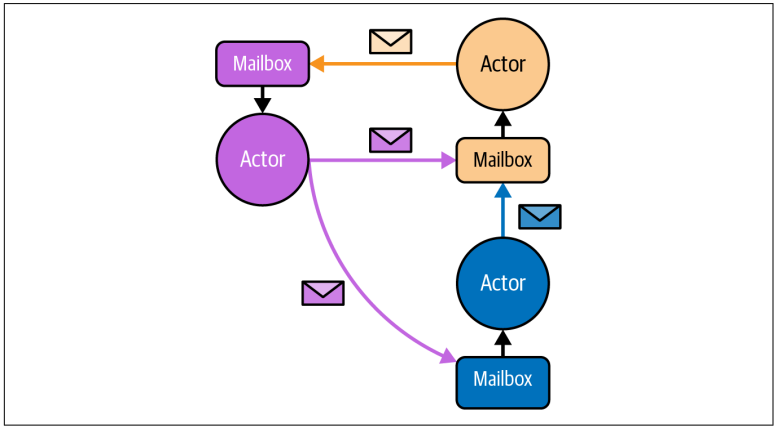


Figure 3-10. A system of actors, communicating with asynchronous messaging

Actors are natural bulkheads (see "Embrace Failure" on page 22). They are isolated and compartmentalized processes that run independently and autonomously. If one actor fails, it does not affect any other actor. Since actors communicate using asynchronous message passing, there are no blocking calls, and as such, no risk of cascading failures. As we have discussed earlier, failures are instead contained and reified as messages, and they can be sent asynchronously to whoever has registered interest. Thus, actors can supervise each other (see "Supervise Subordinates" on page 68).

Actors are so powerful since they inherently embody most of the principles and many of the patterns discussed in this guide, such as asynchronous communication, location transparency, autonomy, decoupling in space and time, virtual addressing, localization of state and processing, consistency boundary, isolated mutations, bulkheading, single responsibility, mobility, decentralization, and message-driven architecture.

Building on these foundational traits, we have in Akka been able to layer higher-level patterns on top. These include replication, sharding, cluster membership, leader election, flow control, stream processing, event sourcing, CQRS, sagas, and more.

Supervise Subordinates

Build self-healing systems using supervisor hierarchies.

As discussed in "Embrace Failure" on page 22, if components are bulkheads with asynchronous boundaries between them, and failures are reified as messages that can be sent to notify interested or dependent components about a particular failure, then one can build systems that heal themselves without external (human) intervention.

Supervisor hierarchies build upon this idea by formalizing it into a general pattern for managing failure that has been used successfully in actor-based languages (like Erlang—which invented it[20]) and platforms (like Akka). Supervisor hierarchies make applications

20 Joe Armstrong's thesis "Making Reliable Distributed Systems in the Presence of Software Errors" is essential reading on the subject. According to Armstrong, Mike Williams at Ericsson Labs came up with the idea of "links" between processes as a way of monitoring process health and life cycle, forming the foundation for process supervision.

extremely resilient to failure by allowing you to build self-healing systems.

Components are organized into supervisor hierarchies in which parent components supervise and manage the life cycle of their subordinate components. A subordinate never tries to manage its own failure; it simply crashes on failure and delegates the failure management to its parent by notifying it of the crash. This model embodies the fail fast philosophy and is sometimes referred to as "let it crash" or "crash-only software."[21] The supervisor then can choose the appropriate failure management strategy, e.g., resume or restart the subordinate, delegate its work, or escalate the failure to its supervisor.

Figure 3-11 presents an example of a supervisor hierarchy.

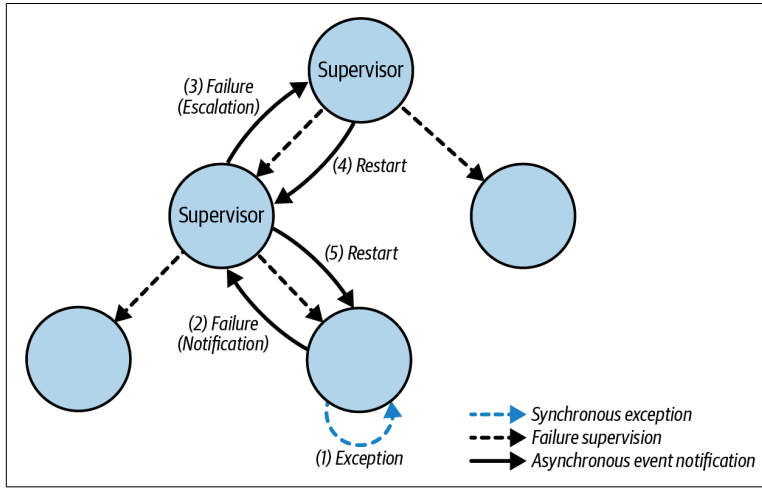


Figure 3-11. A supervisor hierarchy tree of components

In this scenario, one child component fails by raising an exception. The exception is captured by the component itself and reified as a failure message that is sent asynchronously to its supervisor. Upon reception of the message, the supervisor can decide what to do with the failure. In this example, it decides that dealing with this particular failure is beyond its responsibilities, so it escalates the failure up the hierarchy by sending it to its supervisor. The top-level

21 A couple of great, and highly influential, papers on this topic are "Crash-Only Software" and "Recursive Restartability: Turning the Reboot Sledgehammer into a Scalpel", both by George Candea and Armando Fox.

supervisor now decides to restart the entire chain of failed components by sending a restart command, recursively, bringing the whole component hierarchy back to a healthy state, ready to take on new tasks. This is self-healing in action.

With supervision hierarchies, we can design systems with autonomous components that watch out for one another and can recover failures by restarting the failed component(s).

Paired with location transparency (see "Leverage Location Transparency" on page 49)—given that failures are nothing but regular events flowing between the components—we can scale out the failure management model across a cluster of nodes, regions, or clouds while preserving the same semantics and simplicity of the programming model.

To sum up, failures need to be:

- *Contained*, avoiding cascading failures
- *Reified*, as immutable values (facts)
- *Signaled*, asynchronously
- *Observed*, by 1–N separate healthy components
- *Managed*, safely outside failed context

Let's take a vending machine as an example. Imagine we have a vending machine and a customer eager for coffee. The customer can insert a specific amount in coins and get a fresh brew of coffee. If they haven't inserted enough coins, then the customer gets a notification to put in more coins. When enough coins have been inserted, the vending machine returns a cup of coffee (see Figure 3-12).

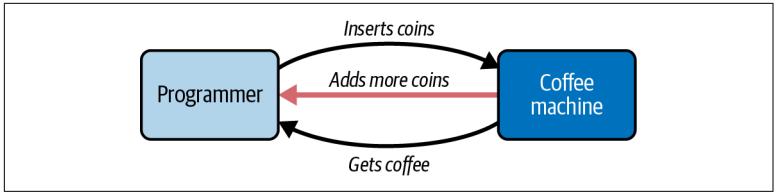


Figure 3-12. Vending machine example: the happy path

Now let's say that the vending machine is out of coffee beans. It could return an error message to the customer saying that it is out of beans. But what should the customer do with that information? It would be better if the vending machine could send this message to the local service technician, who could come out and add more beans (see Figure 3-13). This example is to illustrate a point: in practice, it would of course be useful to let the customer know that the vending machine is out of beans while communicating the problem to the service guy. It's important to define who is responsible for managing the failure versus who (and this can be many entities) can benefit from knowing that it has happened (and that it is being addressed).

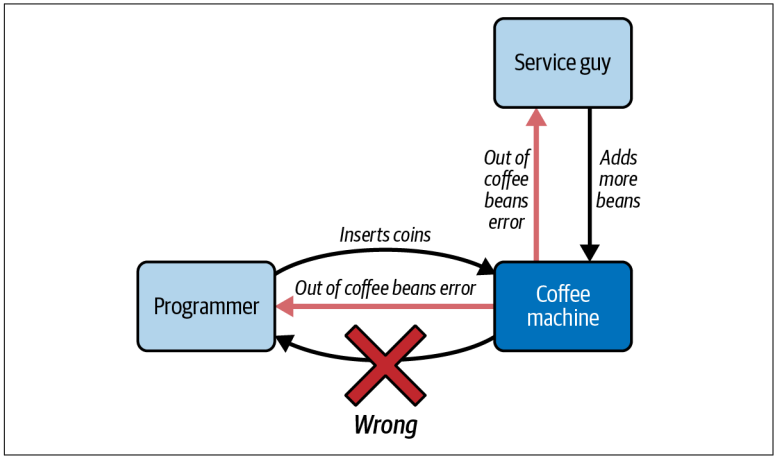


Figure 3-13. Vending machine example: a failure scenario

The same model applies to software components (e.g., actors). Validation errors (such as "insert more coins" in the previous example) should be sent to the user, since those are the user's responsibility and within the user's control to address. But there is little value in sending internal component errors to the user. If each error or failure is reified as a message ready to be sent asynchronously, then other components can subscribe to that message and be notified when it occurs (see Figure 3-14).

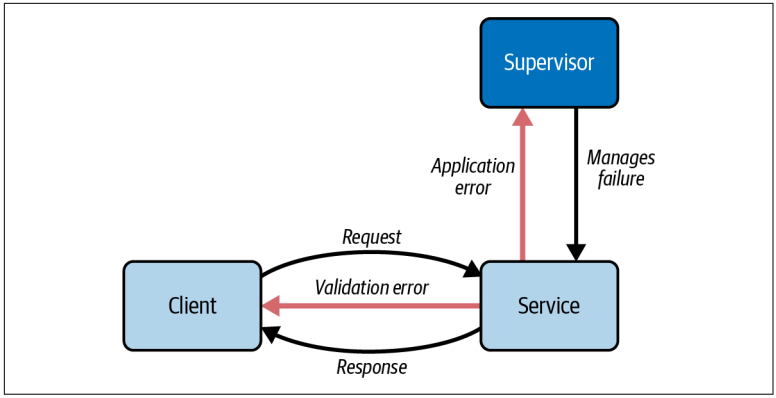


Figure 3-14. Vending machine example: a failure scenario with a failure notification to the supervisor

This allows the messages to be sent up in the hierarchy to the component's supervisor, or to any other interested or dependent component, who is executing in a safe and separate runtime context. Fully decoupled and isolated from the failed component, the supervisor can take appropriate action (e.g., restarting or relocating the component).

Protect Critical Data

Always protect your critical data from failure and uncertainty.

Almost all failures in a distributed system relate to application state/data in some shape or form. The list of what can go wrong with data is a long one. Here are some examples:

Inconsistent data
: Covered extensively in other sections.

Partial data
: You have access only to some of the data you need to make a decision.

Corrupted data
: Data can get corrupted, either on the wire or on disk.

Lost data
: Data can get lost, either on the wire or on disk.

Duplicated data
: Retransmission of data after failure can cause data duplication.

Let's try to classify what we mean by data and their different kinds.

In their very insightful paper "Out of the Tar Pit", Ben Mosely and Peter Marks define *state* as two types of data:

Input data
: This can be essential, since you have to refer to it in the future. It can also be nonessential, only for temporal use, and can be discarded. In the former case, this data is critical and needs to be protected at all costs.

Derived data
: This is not essential since it can always be derived from the input data.

And two types of complexity:

Essential complexity
: The data addresses the essence of the problem being addressed.

Accidental complexity
: All the rest—this is complexity that we would not have to deal with in the ideal world (e.g., complexity arising from infrastructure, performance, elasticity, resilience, etc.).

The paper goes on to define what the ideal system for managing state, architected in three layers (see Figure 3-15):

Essential state
: This is the foundation of the system and completely self-contained. We put our essential input data here. It can make no reference to any other part of the system.

Essential logic
: This is the heart of the system. Often called "business logic," it depends on the essential state but nothing else. Changes in the essential state can require changes in the essential logic.

Accidental state and control
: This is the least important part of the system from a data perspective. *Control* is basically about the order in which things happen. Changes here can never affect essential logic or essential state, but they can be affected by changes in the essential state or essential logic.

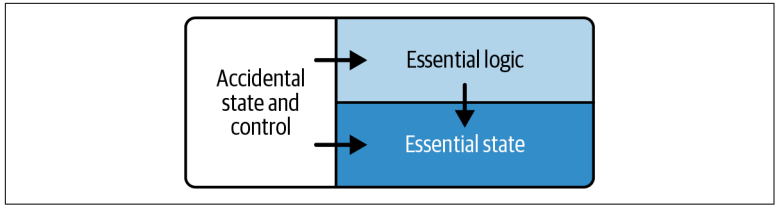


Figure 3-15. Protecting critical state by separating essential logic and accidental logic

How can we put this into action? Leveraging *supervisor hierarchies* (see "Supervise Subordinates" on page 68) gives us an opportunity to create a hierarchy of dependency through which requests flow and add protective layers between the services as a way to protect our *essential state* from failure and uncertainty.

This pattern, called the *error kernel* pattern,[22] describes a design that in some regards mimics the layers of an onion. Instead of having our critical state scattered across the whole application, in different services, we define the minimal error kernel that holds and manages our essential state, that is, our critical "not recomputable" input data or the data that we simply can not lose. Risky behavior is pushed out to more distant layers in the hierarchy, the leaves of the process/service tree. See Figure 3-16 for a schematic of such a design.

The error kernel never performs dangerous operations itself. It never talks to external resources, never accepts user input directly, etc. Instead, it always delegates these tasks to a subordinate process/service. This means that whenever the error kernel is hit with a message, we can assume correctness and safety.

The error kernel pattern builds upon many of the other patterns we have discussed so far: partition state, communicate facts, isolate mutations, go async, bulkheads, detect failure, guard connections, model with actors, and supervise subordinates—all working together in a beautiful orchestration.

22 I can't recommend enough that you read Roland Kuhn's book *Reactive Design Patterns*. It has such a breadth and depth of knowledge about building distributed systems, laid out in a logical and practical way. It, for example, covers the error kernel pattern, among many other patterns.

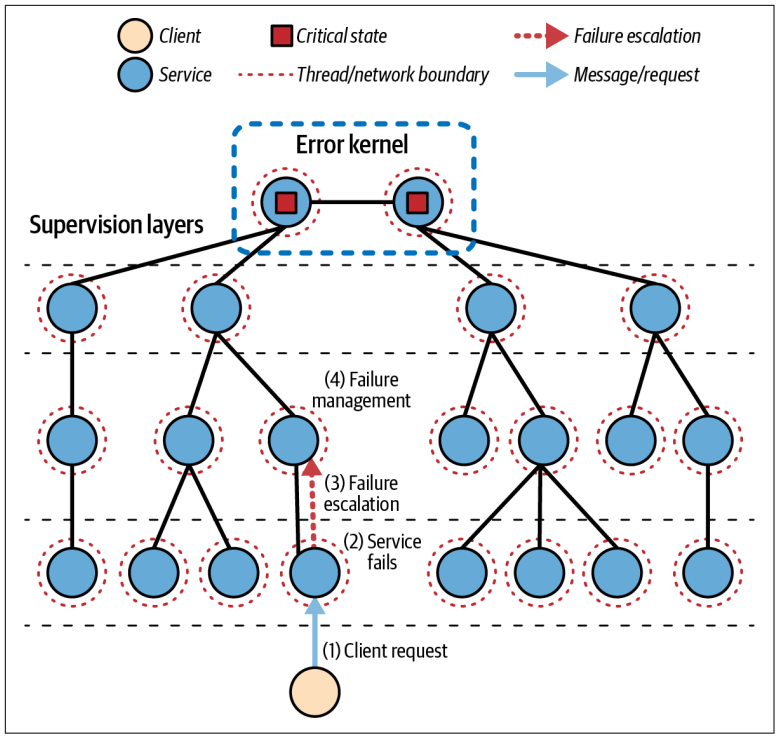


Figure 3-16. The error kernel pattern

Detect Failure

Employ failure detection for more resilient systems.

As we have discussed, a distributed system is a system of discrete parts, each functioning in isolation, bridged by an unreliable network, and trying to function as a single system. Naturally, being able to accurately detect failure to know whether the peer you are communicating with is down is extremely important. But the problem is that there is no perfect failure detector.

Failure detection—first introduced in 1996 by Tushar Deepak Chandra and Sam Toueg in their paper "Unreliable Failure Detectors for Reliable Distributed Systems"—is almost like science mixed with art, where we have to take educated guesses about whether the peer process/node is down or if something else has caused the unresponsiveness. This is easier said than done. Here are a few examples of what can go wrong even if the peer process stays healthy:

- The request is dropped.
- The response is dropped.
- The request is not successfully executed.
- The request is queued up.
- The receiver is struggling with high load or garbage collection.
- The network is congested, causing a delay of the request or response.

A failure detector needs to support two different properties, operating on two discrete axes: *completeness* and *accuracy*. Across this spectrum we have:

Strong completeness
: Every crashed process is eventually suspected by every correct process—*"everyone knows."*

Weak completeness
: Every crashed process is eventually suspected by some correct process—*"someone knows."*

Strong accuracy
: No correct process is suspected ever—*"no false positives."*

Weak accuracy
: Some correct process is never suspected—*"some false positives."*

Ideally, we have strong completeness and strong accuracy. But, as we have discussed previously, every guarantee has a cost. There are many questions to answer and trade-offs to consider. For example, how many false positives can you accept? Or how quickly do you need to detect failures? Lower failure detection latency will cause more false positives, for example, during a garbage collection pause.

In Akka, the heartbeat arrival times are interpreted by an implementation of "The Phi Accrual Failure Detector" by Naohiro Hayashibara and colleagues, in which the suspicion level of failure is represented by a value called *phi*. The basic idea is to express the value of phi on a scale that is dynamically adjusted to reflect current network conditions. It takes network hiccups into account by keeping a history of heartbeat statistics and decouples the monitoring from the interpretation. It can never definitively answer yes or no, but it provides the likelihood—the phi value—that the process is down. The value

of phi is calculated as phi = –log10(1 – *F*(timeSinceLastHeartbeat)), where *F* is the cumulative distribution function of a normal distribution with mean and standard deviation estimated from historical heartbeat inter-arrival times. See Figure 3-17 for a graph with phi plotted against the time since heartbeat.

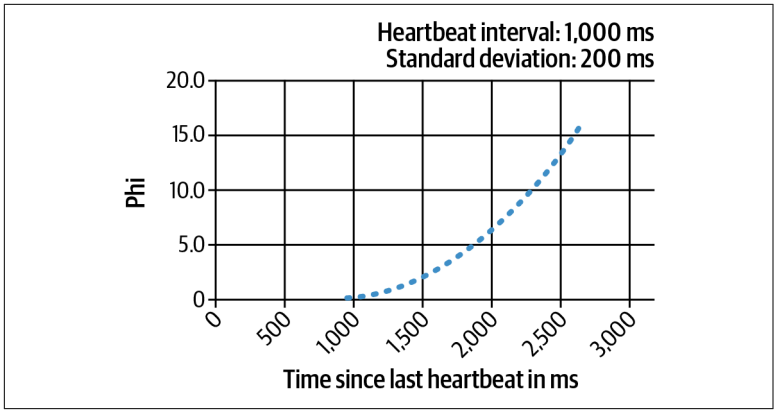


Figure 3-17. Phi versus heartbeat values as calculated by the phi accrual failure detector

Another type of failure detector is discussed in the 2002 paper "SWIM: Scalable Weakly-consistent Infection-style Process Group Membership Protocol" by Abhinandan Das and colleagues. SWIM separates cluster dissemination from heartbeats and introduces the concept of a *quarantine*, where suspected nodes are put. After a time, a grace period window can be marked as faulty. This hybrid algorithm combines failure detection with group membership dissemination (see "Gossip for Convergence" on page 85 for a discussion on group membership and gossip protocols). SWIM uses a clever technique of *delegated heartbeats* to bridge network splits and overcome the problem that occurs when network hiccups cause a node to be unreachable from one node but potentially healthy and reachable from others. The protocol (see Figure 3-18) works as follows:

1. Node N1 picks random member N2 → sends PING(N2).
2. Then either:

   N2 replies with ACK(N2).

   or

   We get a timeout and then:

   a. N1 sends PING(N2) to a set of random members RN.

   b. Each RN sends PING(N2) to N2.

   c. On receipt of ACK(N2), RN forwards it to N1.

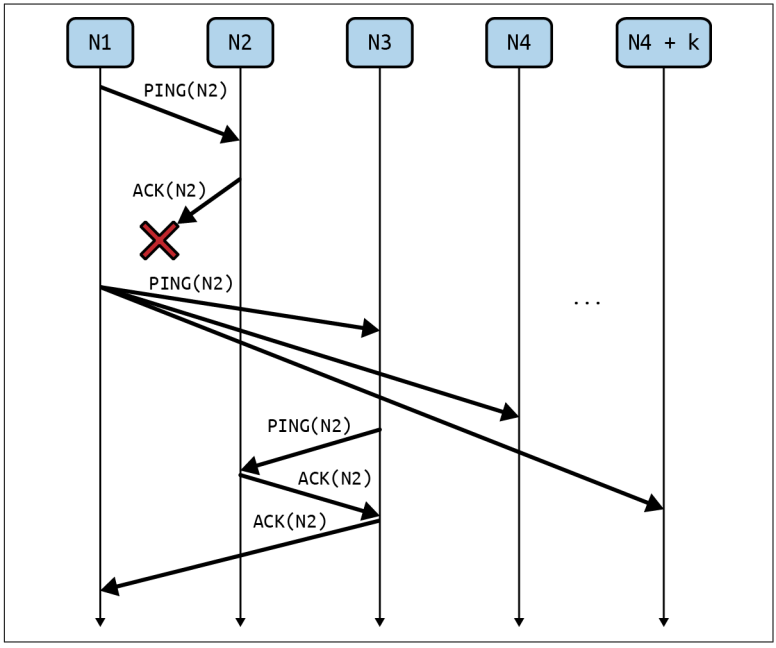


Figure 3-18. Delegated heartbeats in the SWIM failure detector protocol

To summarize, failure detection is critical in distributed systems because it ensures that the system can recognize and respond to failed components, which is essential for maintaining reliability, availability, and consistency. Without accurate failure detection, a system may incorrectly treat a slow or unreachable component as operational, leading to delays, resource lockups, or inconsistencies. Effective failure detection enables the system to reroute requests,

restart processes, or reallocate resources, minimizing downtime and maintaining seamless functionality for users.

Replicate for Resilience

Replicate data for resilience, elasticity, and performance.

Replication[23] is the act of sharing data between multiple discrete parties—services, nodes, data centers—while ensuring consistency as a way to increase availability, resilience, and elasticity:

Synchronous versus asynchronous replication
: We can replicate data *synchronously*, which means that we write the data to the primary storage and the replica simultaneously, in a strongly consistent manner. We can also choose to replicate the data *asynchronously*—decoupling the act of writing to primary storage from the act of sending the data to the replicas. This decoupling has a lot of advantages since it means that we can leverage asynchronous I/O, asynchronous messaging, batching of updates, and decoupling in space and time for better availability, resilience, scalability, and elasticity.

Push-based versus pull-based replication
: We can choose to perform the replication *actively*, by continuously *pushing* data to the replicas as it is written to primary storage. Or we can perform replication *passively*, with the replicas *pulling* data at their own convenience. In passive replication, data can be pulled not just from primary but also from the peer replicas. Passive replication works well with, for example, timeout-based caches.

Sharding is a database-partitioning technique that involves splitting a large dataset into smaller, more manageable pieces, called *shards* (see "Partition State" on page 37). These are distributed across multiple servers. Each shard contains a subset of the total data and can operate independently. This improves scalability and performance by distributing the load and allowing parallel processing. Sharding is often used in distributed systems to handle vast amounts of data and traffic by enabling horizontal scaling, whereby more servers can be added as needed to manage the growing workload (so-called *elasticity*).

23 "A Primer on Database Replication" by Brian Storti is a good overview of (database) replication.

Let's say we have a user dataset indexed by user name that we've sharded across a set of four nodes, with the dataset partitioned into buckets and each node holding a subset of the keyspace (see Figure 3-19).

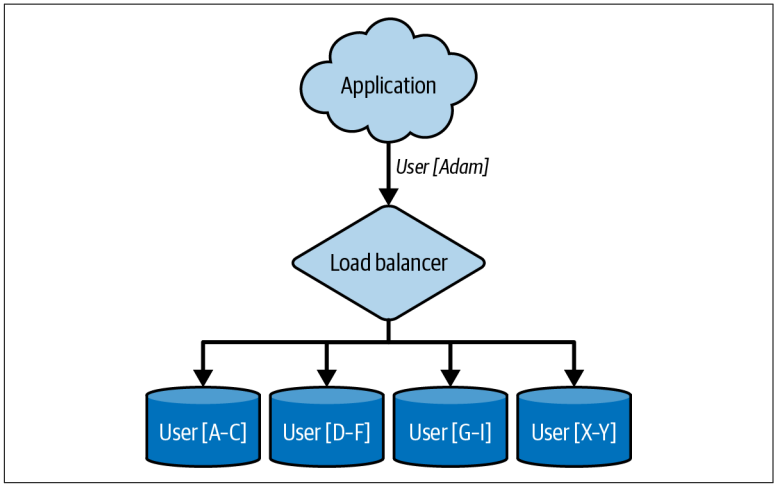


Figure 3-19. Sharding data across multiple different replicas

To ensure redundancy, we can now replicate each partitioned dataset to at least one of the other shard nodes, so that each node holds its own dataset plus backup copies (replicas) of the other nodes' datasets (see Figure 3-20).

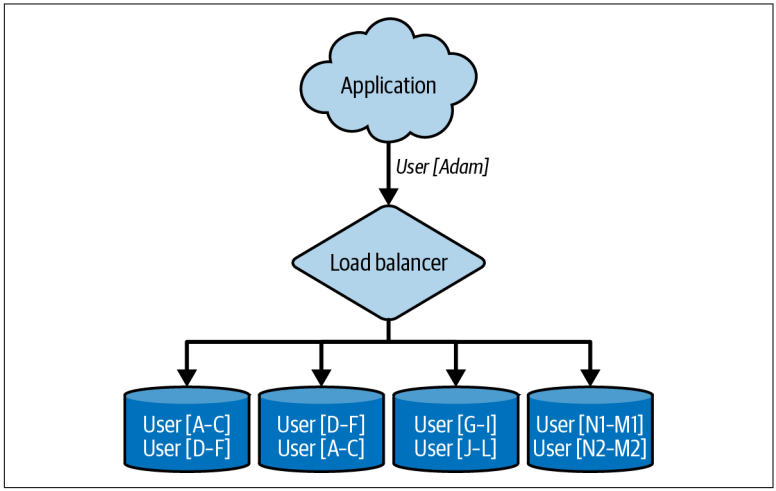


Figure 3-20. Duplicating the sharded data in peer replicas for redundancy

There are different strategies we can use to replicate these datasets between the replicas. Let's now take a look at four of these strategies:

- *Primary–replica (sometimes called active–passive) replication*
- *Primary–primary (sometimes called active–active or multi-master) replication*
- *Buddy replication*
- *Quorum replication*

The simplest strategy is to use *primary–replica replication* (see Figure 3-21). Here we have one node that is the primary node, serving both reads and writes, and multiple (two or more) replica nodes, serving only reads. The primary node is replicating each write to its replica peers.

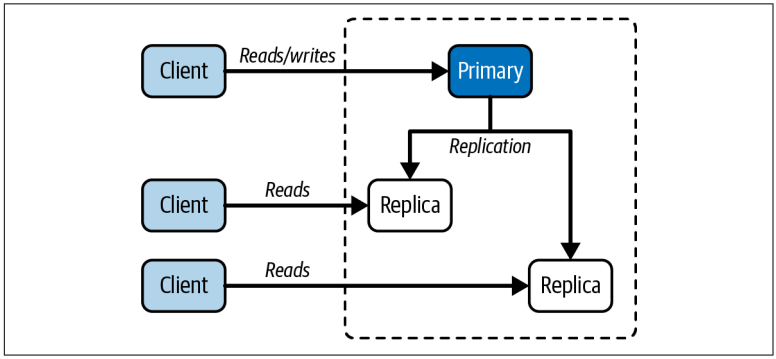


Figure 3-21. Primary–replica replication

A bit more complex but more powerful is *primary–primary replication*. This is also called *multi-master*, which is probably a better term since it hints at the fact that there can be any number of replicas, not just two.

Here we have multiple primary nodes (two or more) that are all serving both reads and writes concurrently (see Figure 3-22). These primary nodes are always "hot," meaning they have the latest data aggregated from all the replicas. Replication is flowing bidirectionally between all replicas. The key here is to manage concurrent updates that result in conflicts, that is, updates in which two or more replicas are updating the same data record.

In practice, there are many ways we can address this issue. Here are some examples:

Last write wins
: This is a simple, brute-force approach with potential data loss, but it can be done automatically without help from the application.

Storing tuples of both of the conflicting writes
: This approach forces the client to reconcile the conflicts afterward (when reading the data), which leaks complexity to the application.

Using CRDTs
: This powerful and flexible approach uses automatic intelligent merging that understands application semantics. It is guaranteed to converge in a consistent fashion but forces the application to model the data as CRDTs, leaking complexity to the application. This is the approach we have taken in Akka.

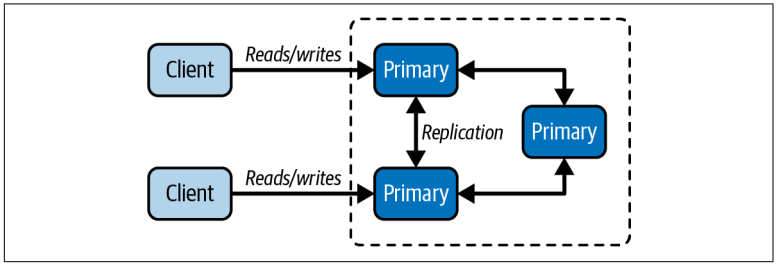


Figure 3-22. Primary–primary/multi-master replication

Another strategy is to use *buddy replication*, which allows you to avoid replicating your data to all instances in a cluster. Instead, each instance picks one or more "buddies" in the cluster and only replicates to these buddies (see Figure 3-23). This strategy is often used in gossip-based systems (see "Gossip for Convergence" on page 85).

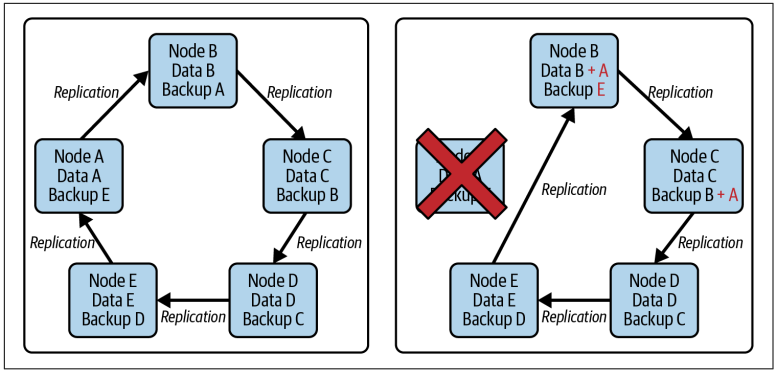


Figure 3-23. Buddy replication

Ideally, we don't want to replicate everything everywhere. That would be both unnecessary and wasteful. The important thing is to think through what guarantees we need for each dataset, noting that these can vary between datasets in the same application (see "Tailor Consistency" on page 27).

We can balance strong and eventual consistency[24] when replicating data by working with three different parameters (using terminology from David Gifford [1979]):

- N = the number of nodes that store replicas of the data.
- W = the number of replicas that need to acknowledge the receipt of the update before the update completes.
- R = the number of replicas that are contacted when a data object is accessed through a read operation.

Using these parameters, we now have the tools for working with a spectrum of consistency guarantees:

- With $W + R > N$ and $W > N/2$, we can guarantee strong consistency.
- $W + R \leq N$ gives us eventual consistency. Note that the system is vulnerable to reading from nodes that have not yet received the updates (a problem that can be addressed with quorum reads, discussed in the next section).

24 Make sure you read Werner Vogels' article "Eventually Consistent—Revisited" for an in-depth discussion of these topics.

- $W < (N + 1)/2$ leaves open the possibility of conflicting writes if the write sets do not overlap.

For some use cases, we don't need to replicate everything to every node in the cluster, as in primary–primary replication. Instead, we can use *quorum replication*,[25] which means that a client update completes only after a quorum of copies has processed it successfully. This can give more flexibility by allowing you to better balance consistency guarantees with traits like availability, scalability, and performance.

Ryan Barrett, in his talk "Transactions Across Datacenters", gives a great overview of the characteristics of different replication and consensus strategies, as shown in Figure 3-24.

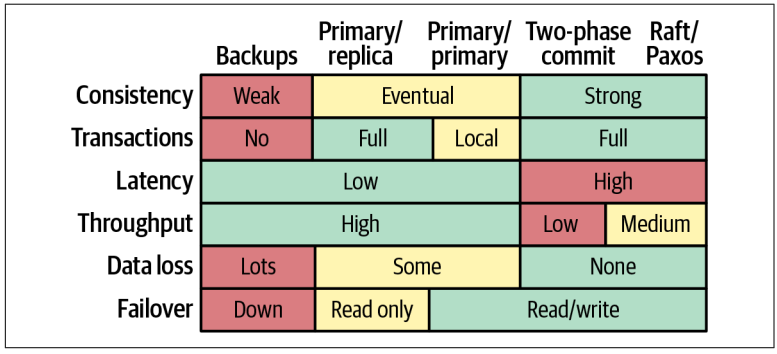


Figure 3-24. Characteristics of different replication and consensus strategies

Replication is key in decentralized systems and works hand in hand with gossip protocols and architectures.

There are many other important questions to discuss related to replication and data consistency (e.g., how to handle data inconsistencies and temporary failures), but they are out of scope for this guide. Here is a short list of links to more information about some techniques that have proven useful:

25 A great paper on eventual consistency and consensus in distributed systems that discusses quorum consensus is the 2013 paper "Rethinking Eventual Consistency" by Philip A. Bernstein and Sudipto Das.

- Hinted handoff
- Strict and sloppy quorum
- Anti-entropy using Merkle trees
- Read repair

Gossip for Convergence

Use epidemic gossiping for elastic cluster membership and data convergence.

We are moving inevitably toward increased decentralization. For most companies, anchored in the cloud and needing to serve their users more efficiently, this means relying less on centralized infrastructure and moving toward decentralized peer-to-peer cloud or cloud-to-edge systems.

Decentralized architecture that distributes logic and data together (see "Localize State" on page 43) throughout a masterless peer-to-peer mesh of nodes offers several advantages for managing data in the cloud and at the edge:

Increased scalability
: With a decentralized architecture, data and workloads can be distributed across multiple nodes, servers, or locations. This allows for more efficient scaling by adding or removing resources as needed, without being limited by the constraints of a centralized system.

Better resilience and high availability
: Decentralized systems are designed to be resilient against single points of failure. If one node or component fails, the system can continue to operate without significant disruption as other nodes can take over the workload. This ensures high availability and minimizes downtime.

Better performance
: Decentralized architectures can perform better by distributing data and workloads across multiple nodes, reducing bottlenecks, and leveraging parallel processing capabilities. This benefits data-intensive workloads like big data analytics or AI and machine learning applications.

Improved data locality
: In a decentralized architecture, data can be stored and processed closer to where it's generated or consumed. This can reduce network latency and improve overall performance, especially for large data transfers or in real-time data processing scenarios.

Enhanced security and data privacy
: Decentralized architectures can help mitigate security risks by distributing data across multiple nodes or locations. This makes it more difficult for unauthorized parties to access or compromise the entire dataset, as they would have to breach multiple nodes simultaneously. It also helps to ensure that data is contained in a geographic region (e.g., for compliance reasons).

Cost optimization
: Decentralized architectures can help optimize data management and processing costs by leveraging infrastructure resources more efficiently and scaling resources dynamically.

Reduced risk for vendor lock-in
: Decentralized architectures often rely on open source technologies, standardized interfaces, multi-cloud, and edge infrastructure. Thus, they reduce the risk of vendor lock-in and allow for greater flexibility in choosing cloud providers or switching between them.

Decentralized clustering involves multiple nodes working as a single entity. By collaborating among themselves and sharing a single endpoint, they enable the elasticity and resilience of the service. The question is: how can we make multiple nodes act as a single entity?

A key problem is how to disseminate information in a cluster, automatically and seamlessly. Many systems rely on *gossip protocols*, or *epidemic protocols*.[26] This class of decentralized communication protocols is used to efficiently spread information, achieve consensus, or keep consistency among a large number of nodes. These protocols mimic the way gossip spreads in social networks or viruses spread in biological systems, with information spread randomly between pairs of nodes over time until it reaches the whole network.

26 An interesting paper on how to efficiently implement gossip protocols is "Efficient Reconciliation and Flow Control for Anti-Entropy Protocols" by Robbert van Renesse, Dan Dumitriu, Valient Gough, and Chris Thomas.

There are two classes of gossip: anti-entropy and rumor-mongering protocols. *Anti-entropy protocols* gossip information until it is made obsolete by newer information; they are useful for reliably sharing information throughout a cluster of nodes. *Rumor-mongering protocols* have participants gossip information for some amount of time that is sufficient to ensure a high likelihood that all nodes in the cluster have received the information. The key features of gossip protocols are:

Decentralization
: Gossip protocols do not rely on a central authority; every node in the network has equal responsibility for propagating information.

Scalability
: Gossip protocols are highly scalable; by handling large numbers of nodes efficiently due to their probabilistic nature, they ensure that the load is evenly distributed.

Fault tolerance
: Gossip protocols are robust against node failures, as the redundant dissemination of information ensures that even if some nodes fail, the information eventually reaches all nodes.

Convergence
: Over time, gossip protocols ensure that all nodes in the network reach a consensus or have consistent data, though exact guarantees may depend on the specific implementation.

Gossip protocols make it straightforward to implement the following:

Cluster membership[27]
: The membership information is gossiped, allowing nodes to dynamically and seamlessly join and leave the cluster or become temporarily unavailable, without disrupting the cluster. In Akka, we are using an optimized style of gossip called push/pull gossip.

27 An insightful paper worth reading discussing cluster membership and failure detection in gossip protocols is "SWIM: Scalable Weakly-consistent Infection-style Process Group Membership Protocol" by Abhinandan Das, Indranil Gupta, and Ashish Motivala.

Failure detection
: Heartbeat messages are gossiped[28] at a predefined temporal cadence, allowing anomalies to be detected (see "Detect Failure" on page 75).

Leader election
: Information about which node is considered the current leader (or has a specific role) is gossiped. In Akka, this actually does not happen through an "election" but is deterministically decided (the leader is simply the first node in a sorted node ring) after so-called "cluster convergence" has occurred (convergence can be defined as the arrival at local proof of a global state in the past when locally a gossip is observed where all nodes are represented in the "seen set"; see Akka documentation for details).

Decentralized architecture is based on the idea that there are no special nodes, no leaders or masters, but all nodes are equal and communicate with each other in a peer-to-peer fashion. *Peer-to-peer systems* are generally based on using a "semantic-free index," which means that each data item that is to be maintained by the system is uniquely associated with a key that can be used as an index.

This allows the storing of key–value tuples. Each node in the cluster is assigned an identifier from the same set of all possible hash values, and each node is made responsible for storing data associated with a specific subset of keys. Pairing this with an efficient function for looking up each data item based on its key means that such a system can be seen as implementing a *distributed hash table* (DHT).

The topology of such a system is crucial since each node can be asked to look up a specific key and may need to, in the case of a "lookup miss," have an efficient way of routing the request to the node holding the key and its data. The Chord[29] system was the first to suggest such a solution, and it guarantees finding the data in $O(\log(N))$ jumps. It lays out all the nodes in the cluster logically in a so-called "node ring" (see Figure 3-25).

28 See "A Gossip-Style Failure Detection Service" by Robbert van Renesse, Yaron Minsky, and Mark Hayden.

29 See the 2001 paper on "Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications" by Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan.

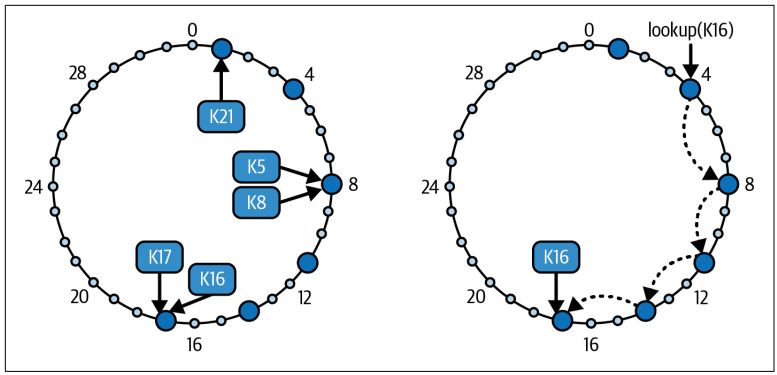


Figure 3-25. Chord node membership ring—finding data in log(N) jumps

Using these techniques, application data can be put in buckets and sharded (see "Replicate for Resilience" on page 79) across all the (currently) available nodes, allowing the cluster to be fully elastic and responsive to changes in workload.

The most common technique to do this evenly and allow for dynamic resizing of the cluster is *consistent hashing*.[30] In this technique, we evenly distribute K objects across N buckets (roughly about K/N for each). When the number of buckets (N) changes, we don't need to move/rebalance all objects but only K/N of them. To do this, we use a base hash function, such as SHA-1. This gives minimal disruption to managing the dataset when nodes join or leave the cluster.

The output of the hash function hash(key) = P is a range 0...m – 1. This represents our hash ring where:

- key = the bucket ID.
- P = the position on the hash ring.
- m = the total range of the hash ring.

30 Consistent hashing was first defined in "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web" by David Karger and colleagues.

We hash the data buckets—the keys for the buckets—using the hash function and distribute them out on the node ring based on the output of the hash values. Then we also hash the incoming requests—the extracted keys for the data buckets—using the same hash function and distribute them out on the ring based on the hash.

To allow for availability and consistency, these data buckets also need to be replicated to more than one node in the node ring. In this way, if one or more nodes are lost, the system can still serve requests by simply redirecting the request to one of the available replica nodes. This can also allow for a higher level of consistency guarantees (remember that we are gossiping the data throughout the cluster in an eventually consistent fashion, see "Tailor Consistency" on page 27) by always reading data from more than one replica and ensuring consistency of the data before returning it to the user. Most often, a quorum read based on a *replication factor*—usually an odd number (e.g., 3, 5, or 7 nodes)—is used. See Figure 3-26.

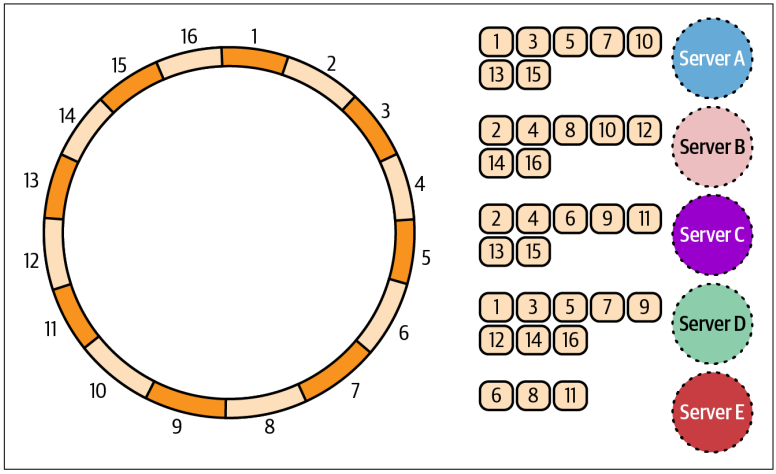


Figure 3-26. Data is spread out and duplicated in buckets across the node/hash ring

Gossiping can be inefficient if there are a large number of nodes in a cluster. A couple of techniques that can be used to mitigate this problem are separation of failure detection heartbeat and dissemination of data, as done in the SWIM protocol (discussed in "Detect Failure" on page 75), and *push/pull gossip*, discussed in the 2003 paper "Push-Pull Gossiping for Information Sharing in Peer-to-Peer Communities" by Khambatti and colleagues. Push/pull gossip is, for example, used in Akka.

Many products and systems have used these techniques successfully over the years, among others, Amazon's Dynamo,[31] Apache Cassandra, and Akka Cluster. For a high-level summary of how Akka Cluster works, see "Akka Cluster Implementation Notes". Among other things, it uses Dynamo-style gossiping, biased gossip, push/pull gossip, CRDTs for gossip state, and leader election.

Seek Consensus

Leverage consensus algorithms for coordination and consensus.

We have talked a lot about techniques and patterns for leveraging the inherent nature of eventual and causal consistency for better availability, scalability, and elasticity. But sometimes we have to ensure consensus across discrete components in a strongly consistent way. It's time to discuss consensus algorithms.

In a fully asynchronous message-passing distributed system where at least one process can fail, the 1985 classic Fischer, Lynch, and Paterson (FLP) impossibility result proves that a deterministic consensus algorithm is impossible. This result is based on worst-case scheduling scenarios, which are rare in practice except in adversarial cases like a targeted denial-of-service attack. Typically, process scheduling involves some natural randomness.

31 The Dynamo paper is essential reading since it is packed with interesting concepts and has inspired so many NoSQL databases: "Dynamo: Amazon's Highly Available Key–Value Store".

The FLP impossibility result shows that in a fully asynchronous distributed system with even one potential process failure, it's impossible to guarantee that a deterministic consensus algorithm will always terminate and reach a consensus. This impossibility arises because, in such systems, unpredictable delays and failures can prevent processes from ever agreeing, especially in worst-case scenarios.

What do we require from a consensus protocol? There are four important properties that we care about:

Termination
: Every process eventually decides on a value *V*.

Validity
: If a process decides *V*, then *V* was proposed by some process.

Integrity
: No process decides twice.

Agreement
: No two correct processes decide differently.

Consensus algorithms like Paxos[32] and Raft are designed with the FLP result in mind. They acknowledge the impossibility of guaranteeing consensus in every possible scenario, especially under the conditions described by FLP. However, these algorithms use specific techniques to achieve consensus in practice, under typical conditions where worst-case scenarios are rare. Other consensus algorithms include the Zab[33] protocol used by Zookeeper and Viewstamped Replication,[34] which predates Paxos and is similar to Raft in some regards.

We will now focus on Paxos and Raft since they are the most used protocols in the industry today:

32 A good read on how Paxos works is the paper "Paxos Made Moderately Complex" by Robbert Van Renesse and Deniz Altinbuken.

33 For details on Zab, read the 2008 paper "A Simple Totally Ordered Broadcast Protocol" by Benjamin Reed and Flavio P. Junqueira.

34 For details on viewstamped replication, read the 1988 paper "Viewstamped Replication: A New Primary Copy Method to Support Highly Available Distributed Systems" by Brian M. Oki and Barbara H. Liskov.

Paxos
: Designed by Leslie Lamport, Paxos achieves consensus by breaking the process into phases with quorum-based voting, allowing it to make progress even if some processes fail or messages are delayed. While it doesn't guarantee consensus in every theoretical scenario (as per FLP), it works effectively under most practical conditions.

Raft
: Designed by Diego Ongaro and John Ousterhout, Raft simplifies the consensus process compared to Paxos, making it easier to understand and implement. It uses leader election, log replication, and a strong leader to coordinate decisions, helping to avoid the complexities that arise from the FLP impossibility in practice. Raft is designed to work efficiently in real-world distributed systems, even if it can't overcome the FLP limit in every case.

There are two categories of consensus protocols:

Single-value consensus protocols
: These focus on getting nodes to agree on a single value, often encoding metadata such as a database transaction. Binary consensus, a subset of this, limits the value to a binary choice {0,1} and serves as a building block for more complex protocols. Includes protocols such as Paxos.

Multivalued consensus protocols
: These extend the concept of single-value consensus by agreeing on a sequence of values over time, creating a growing history. While this could be done by repeatedly running single-value consensus, multivalued protocols include optimizations for greater efficiency and support for tasks like reconfiguration. Includes protocols such as Multi-Paxos and Raft.

Let's take a brief look at Raft since it is probably the simplest and most popular consensus algorithm today. Raft operates in a cluster of servers, where one server acts as the leader and the others as followers. The leader is responsible for managing the replicated log, which stores a sequence of commands that are applied to the system's state. Followers passively replicate the log entries from the leader and only respond to its requests.

The key components of Raft are:

Leader election
: Raft begins with a leader election process. If a server doesn't receive communication from a current leader within a timeout period, it assumes the leader has failed and starts an election by becoming a candidate. The candidate requests votes from other servers, and if it gains a majority, it becomes the new leader. This process ensures that the system can quickly recover from leader failures. See Figure 3-27 for an overview of the different steps for leader election in the Raft protocol.

Log replication
: Once a leader is elected, it starts accepting client requests. The leader adds each new command to its log and then replicates these entries to its followers. The leader waits for a majority of followers to acknowledge the entry before considering it committed. Once committed, the leader applies the entry to its state machine and tells the followers to do the same.

Safety
: Raft ensures that all servers eventually agree on the same log entries in the same order, even if some servers crash and recover. It achieves this by ensuring that leaders only commit entries that are safely replicated to a majority of servers, preventing any conflicting entries from becoming committed.

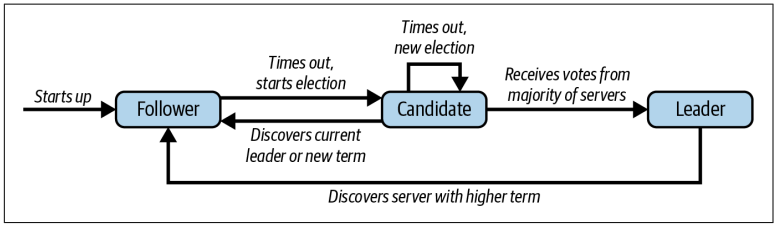


Figure 3-27. State-transition diagram for Raft leader election

Raft also supports dynamic reconfiguration, allowing the set of servers in the cluster to change without disrupting the consensus process. Additionally, Raft includes optimizations like log compaction to improve performance over time, ensuring that the log doesn't grow indefinitely.

Both Paxos and Raft are examples of how consensus algorithms can be designed to perform well in practice, despite the theoretical limitations highlighted by the FLP result. They accept that consensus might not be achievable in the worst case but still manage to achieve it reliably in realistic scenarios.

Consensus protocols like Paxos and Raft are crucial in distributed systems because they ensure consistency across multiple nodes, even in the presence of failures. In a distributed system, nodes must agree on decisions—such as updating shared state or completing a transaction—to avoid conflicts and ensure reliable operation. Consensus protocols help nodes agree on a single source of truth, allowing them to achieve consistency without centralized control.

Let's look at a simple example. Imagine a distributed banking system where several nodes process transactions. If two users simultaneously attempt to withdraw the last $100 from the same account on different nodes, there must be a mechanism to prevent double spending. A consensus protocol like Raft helps nodes reach agreement on who gets the funds first by coordinating to finalize each transaction's order, ensuring that the account balance is updated consistently across all nodes. Without consensus, the system risks conflicting updates, leading to data inconsistencies.

Now that we have learned patterns that can help us build distributed systems that leverage cloud and edge infrastructure in a natural and optimal way, let's take some time to discuss how we can design applications, in particular event-driven and data-driven microservices, that make use of these foundational principles and patterns.

CHAPTER 4

Designing Distributed Applications

Now that we have delved into the principles and patterns of distributed application architecture, how can we best put them into practice? Using event-driven architecture with microservices has proven to be one of the best ways to translate business needs and model the application domain in a way that is in line with how distributed systems work, allowing applications to be ready for, and make the most out of, the cloud and edge. Let's now take some time to discuss this in more detail.

Fundamental Principles of Microservices

Microservices have emerged as a powerful architectural pattern for building cloud applications, offering numerous advantages in terms of speed, efficiency, predictability, and safety. This approach has proven particularly effective in scaling organizations by allowing developers to decompose complex applications into discrete, decoupled services that communicate via well-defined protocols. This decomposition empowers individual teams to develop, deploy, and evolve their microservices independently, fostering agility and innovation.

Moreover, when implemented correctly, microservices provide an elegant solution to the challenges inherent in distributed systems. To harness the full potential of this architecture, you should adhere to five key principles:

- Isolation of components
- Autonomous operation

- Single responsibility
- Exclusive state ownership
- Mobility with addressability

Let's explore each of these principles in greater detail.

Isolation of Components

Isolation (see "Isolate Mutations" on page 40 and "Localize State" on page 43) is the most important trait and the cornerstone of many high-level benefits in microservices. This principle has the biggest impact on your design and architecture. It will, and should, slice up the entire architecture, so you need to consider it from day one.

Isolation even affects how you break up and organize teams and their responsibilities. This relates to Melvin Conway's 1967 discovery, later named Conway's law: "Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure."

Isolation is fundamental to microservice design because it allows each service to operate independently while reducing risk and complexity at various levels of the system. Different elements can be isolated:

Code isolation
: Each microservice should have its codebase isolated to create a boundary for state and logic. This isolation allows services to function as "bulkheads" in the system (see "Embrace Failure" on page 22 and "Isolate Mutations" on page 40); if one service fails, it doesn't bring down others. Code isolation also supports Agile development practices, enabling teams to develop, debug, and release each service independently, often through dedicated continuous integration and continuous delivery (CI/CD) pipelines tailored to the service's unique requirements.

Packaging isolation with containers
: Containers provide a consistent runtime environment that bundles the code, dependencies, and configuration for each microservice, isolating each one from others at the operating system level. Containers allow services to run independently, avoiding conflicts in dependencies and environmental configurations.

This also facilitates portability, making it easy to deploy services in different environments without modification.

Compute isolation with pods
: In systems like Kubernetes, *pods* are units of compute that encapsulate one or more containers. They help provide resource isolation at the infrastructure level, ensuring each microservice has its allocated CPU, memory, and network resources. This prevents a resource-intensive service from starving others, which is particularly important in large-scale, multi-tenant cloud environments.

API and event isolation
: Microservices communicate through APIs or asynchronous messaging (see "Go Async" on page 44), which provides a clean separation between the service's internal workings and its external interactions. By isolating inputs and outputs, services are more resilient to changes and failures; they can gracefully handle issues like timeouts or retries without leaking problems back to the client (see "Guard Connections" on page 62). This isolation is key for supporting patterns like eventual consistency (see "Tailor Consistency" on page 27), which is essential in distributed systems.

Database isolation
: Isolating databases, by ensuring a separate database per service, prevents services from inadvertently affecting each other's data. This reduces the risk of data corruption and makes it easier to manage schema changes without impacting other services. Each service can also choose the storage mechanism that best fits its requirements, enhancing flexibility and performance.

Operational and failure isolation
: Isolation at the operational level includes metrics, logging, and monitoring, all scoped to individual services (see "Observe Dynamics" on page 47). This separation makes it easier to detect and troubleshoot issues since failures or slowdowns are confined within a single service. Furthermore, isolating fault domains through circuit breakers (see "Guard Connections" on page 62) and load balancing (see "Replicate for Resilience" on page 79) ensures that issues in one service don't cascade throughout the system, enhancing system resilience.

By enforcing isolation across these dimensions, microservices can evolve independently, scale as needed, and reduce the risk of system-wide failures. This separation ultimately allows cloud applications to be more agile, resilient, and elastic.

Autonomous Operation

Isolation is a prerequisite for *autonomy* (see "Assert Autonomy" on page 25). Only when services are truly isolated can they achieve full autonomy, making decisions independently, acting on their own, and cooperating with other services to solve problems.

Mark Burgess captures this concept well in his work on promise theory: "With a promise model, and a network of autonomous systems, each agent is only concerned with assertions about its own policy; no external agent can tell it what to do, without its consent."[1]

Working with autonomous services provides flexibility in several areas: service orchestration, workflow management, collaborative behavior, scalability, availability, and runtime management. However, this flexibility comes at the cost of requiring more thoughtful design of well-defined and composable APIs.

Autonomy extends beyond just system architecture and design. A system built with autonomous services allows the teams developing these services to maintain their own autonomy. They can roll out new services and features independently, fostering agility and innovation.

Ultimately, autonomy serves as the foundation for scaling both the system and the development organization behind it. It enables growth and adaptability at both the technical and organizational levels.

Single Responsibility

The Unix philosophy[2] and design, despite their age, remain highly effective and relevant. One of the core tenets is that developers should create programs with a single, well-defined purpose.

1 Paul Borrill, Mark Burgess, Todd Craw, and Mike Dvorkin, "A Promise Theory Perspective on Data Networks", 2014.

2 The Unix philosophy is described really well in the classic book *The Art of Unix Programming* by Eric Steven Raymond.

These programs should be designed to work seamlessly with other small programs through good composition.

This concept was later introduced to the object-oriented programming community by software engineer Robert C. Martin, who termed it the single responsibility principle (SRP).[3] SRP states that a class or component should "have only one reason to change."

Much debate has centered around the ideal size of a microservice. Questions like "What qualifies as 'micro'?" or "How many lines of code can a microservice have?" miss the point. Instead, *micro* should refer to the *scope of responsibility*. The guiding principle here is that of the Unix philosophy and SRP: *a service should do one thing and do it well*.

When a service has a single reason to exist, *providing a single composable piece of functionality*, it prevents the entanglement of business domains and responsibilities. This approach makes each service more generally useful, and the overall system becomes easier to scale, more resilient, simpler to understand, and easier to extend and maintain.

As an example, let's take a look at the design in Figure 4-1. Here we have a simple ordering system with three services: orders, payment, and inventory, each providing a single well-defined piece of functionality while working together. The services are encapsulated by *bounded contexts*,[4] which establish clear boundaries for services. A bounded context acts as a complexity bulkhead, keeping unnecessary complexity from leaking out while maintaining a unified domain model and language within.

3 For an in-depth discussion of the single responsibility principle, see Robert C. Martin's website "The Principles of OOD [Object-Oriented Design]".

4 For an in-depth discussion of how to design and use bounded contexts, read Vaughn Vernon's book *Implementing Domain-Driven Design* (Upper Saddle River, NJ: Addison-Wesley, 2013).

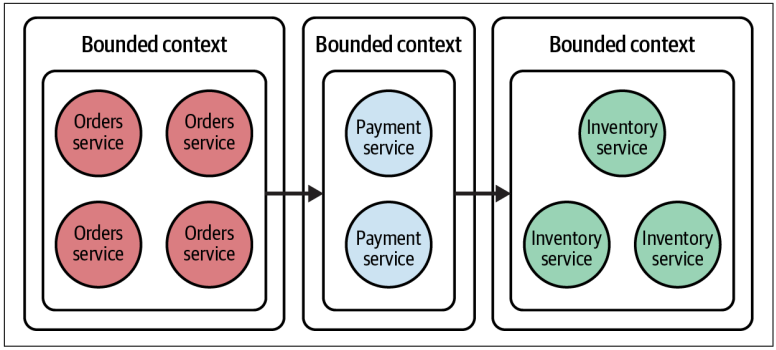


Figure 4-1. An ordering system comprised of three services that each provide a single well-defined piece of functionality

Exclusive State Ownership

Up to this point, we've characterized microservices as a set of isolated services, each with a single area of responsibility. This framework allows us to treat each service as an independent unit that can live, die, and move in isolation—a crucial factor for both resilience and elasticity. *Elasticity* here refers to a system's ability to adapt to changing input rates by adjusting the resources allocated to handle these inputs.

While this sounds promising, we must address a critical factor: *state*.

Microservices are typically stateful components, encapsulating both state and behavior. Importantly, the principle of isolation extends to state as well, necessitating that we treat state and behavior as a unified entity.

Services need to *own their state exclusively* (see "Partition State" on page 37). This simple fact has profound implications. It means that data can only be strongly consistent *within* each service, not *between* services. For interservice data consistency, we must rely on eventual consistency and forgo traditional transactional semantics.

This approach requires abandoning the concept of a single, all-encompassing database. It means moving away from fully normalized data and cross-service joins (as illustrated in Figure 4-2). We're entering a different paradigm that demands a new way of thinking, along with different designs and tools. We'll explore these concepts in greater depth later.

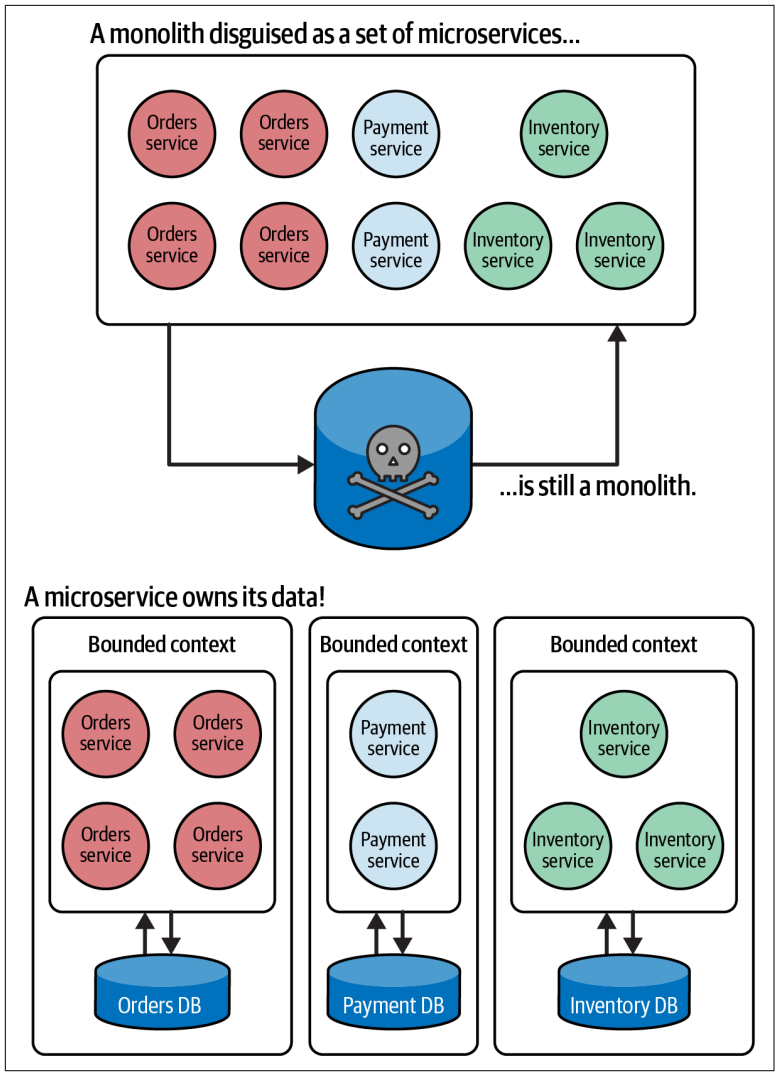


Figure 4-2. A monolith disguised as a set of microservices

Mobility with Addressability

With cloud computing, virtualization, and Docker containers, we have powerful tools to manage hardware resources efficiently. However, these benefits are lost if our microservices and underlying platforms are locked into a fixed topology or deployment setup. What we need are mobile services that allow the system to be elastic and adapt dynamically to usage patterns.

Mobility means you can move services around during runtime, and pairing mobility with location transparency (see "Leverage Location Transparency" on page 49) ensures they remain agnostic to the system's current deployment or topology—both of which should be able to change dynamically. Mobility also allows services to be moved or reallocated without disrupting their functionality or client interactions. Here are some examples of situations in which this is essential:

Scaling and load balancing
: When a service experiences high demand, new instances may be deployed dynamically across different nodes or even different regions. For example, during a seasonal sales event, an ecommerce application might scale its checkout and inventory services. Location transparency enables load balancers or service discovery mechanisms to redirect traffic seamlessly to the new instances without the client needing to know their specific locations.

Failover and disaster recovery
: In the event of node failure, services might need to be relocated to another server or data center. A location-transparent service can be restarted on a different node, and peer services or clients can reconnect without awareness of the shift. For instance, a banking application might rely on disaster recovery to reroute requests to a backup instance of a critical service in another region during a regional outage.

Edge computing and proximity optimization
: Sometimes, services are moved closer to the data source or the user to reduce latency. For example, in city traffic monitoring, services may need to migrate between edge nodes to handle local load spikes or failures while maintaining low-latency, real-time data for emergency response. Location transparency enables seamless access to this data, ensuring uninterrupted service despite node mobility.

Cluster maintenance and upgrades
: Routine maintenance or infrastructure upgrades often require services to be moved. A Kubernetes cluster, for example, might reschedule pods onto different nodes to apply updates. Here, location transparency ensures that service consumers don't experience disruption.

Cost optimization
: Cloud providers often move workloads dynamically to optimize resource usage across regions or availability zones (AZs). Mobile and location-transparent services can be moved to cheaper instances or regions automatically, which can reduce costs in the long run.

In all these cases, mobility and location transparency ensure that clients and other services don't have to be updated with each change in service location, making the system more resilient, more cost-effective, and easier to manage at scale.

Now that we've outlined the five key traits of a microservice, we can begin to dismantle the monolith and apply these traits in practice.

Slaying the Monolith

Before tackling the task of "slaying the monolith," it's important to understand why this architecture is problematic and why a shift to decoupled microservices is necessary.

Consider a monolithic Java application with a typical three-tier architecture, using technologies like servlets, EJB, Java Persistence API (JPA), and a SQL database (or the Spring Framework for a modern version). Figure 4-3 depicts such an application.

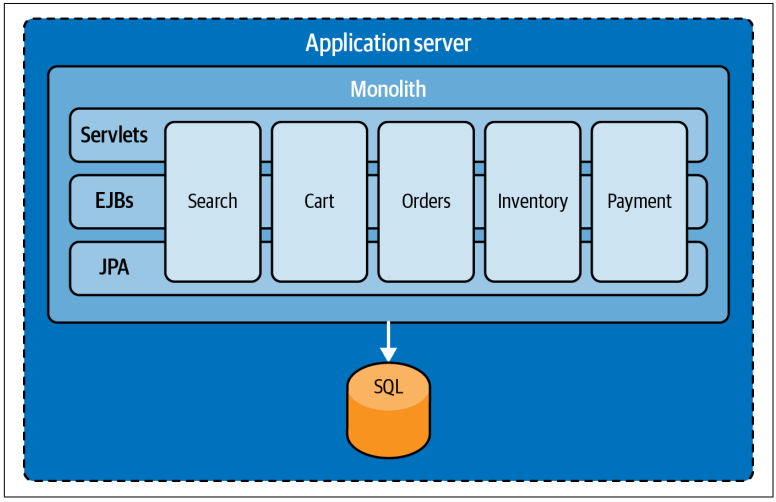


Figure 4-3. A monolithic application with a classic three-tier architecture

The issue with this design is that it creates strong coupling between components within each service and across services. Complex workflows driven by deep synchronous call chains entangle services, making the system difficult to understand and preventing services from evolving independently. Each service holds the caller hostage until all methods execute, creating rigid dependencies.

This tight coupling means that you must upgrade all services together, and failures are hard to isolate. A failure in one service can cascade through the tiers, potentially bringing down the entire application. Exception handling through "try–catch" statements is a blunt tool that further complicates failure management. Moreover, the lack of service isolation makes it impossible to scale individual services; even if only one service experiences high traffic, you must scale the entire monolith, leading to inefficiency.

Traditional application servers reinforce this monolithic approach, bundling the services and deploying them into a single instance. The application server then manages the service "isolation" through class loader magic. This fragile model forces services to compete for resources like CPU, memory, and storage, reducing fairness and system stability.

When refactoring a monolith into a microservices-based system, many developers take the path of least resistance—using scaffolding tools, containers, and synchronous APIs. This often results in an architecture like the one in Figure 4-4, where services are single instance, communicate over synchronous RPC, and rely on CRUD operations through JPA with a SQL database (sometimes still a monolithic, shared one). What's been created is what I call microliths—essentially small monoliths.

Avoid Microliths

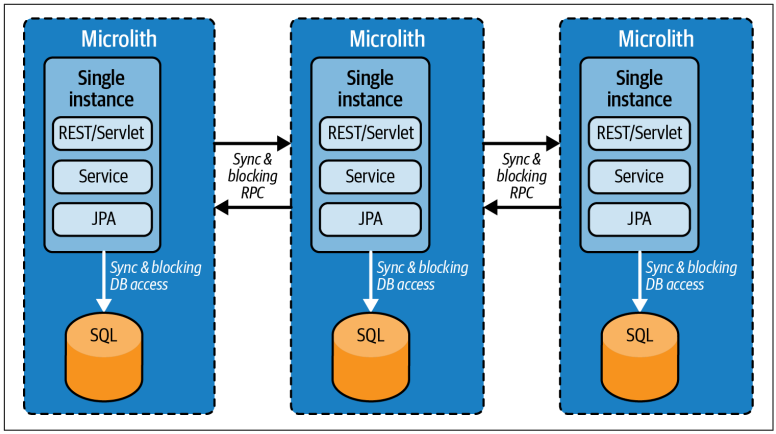


Figure 4-4. A system of microliths communicating over synchronous protocols

A *microlith* is a single-instance service that replaces synchronous method calls with synchronous and blocking RPC, while still using blocking database access. This preserves the strong coupling of the original monolith but introduces the added overhead of synchronous network calls, increasing latency. Since each service is a single instance, it inherently lacks scalability and availability. A single instance can't scale out or stay available if it fails.

Some might assume that using Docker or Linux (LXC) containers solves this, but merely packaging microservices into containers doesn't address the core issues. Containers, and orchestration tools like Kubernetes, are excellent for managing and scaling instances, but they don't solve the fundamental challenges of distributed systems (as discussed in Chapter 1).

As we have discussed throughout this guide, real complexity lies in the *space in between* services—in areas like communication, coordination, consistency, and managing shared state and resources. These are intrinsic to the design of the application itself and can't be easily added afterward.

Microservices Come in Systems

"One actor is no actor. Actors come in systems." This wise statement is often attributed to Carl Hewitt, creator of the actor model. The same applies to microservices: "One microservice is no microservice. Microservices come in systems."

Like humans, microservices are autonomous and must communicate and collaborate with others to solve problems. It's in this collaboration where the most interesting opportunities and toughest challenges arise.

The real difficulty in microservices design isn't in building the individual services—it's managing the *space in between* them. Designing effective microservice systems requires a deep understanding of how services interact, coordinate, and share responsibilities to work together seamlessly.

We are forced to embrace uncertainty, failure, autonomy, decoupling, asynchronous messaging, and data partitioning and be ready to give up some of the consistency guarantees, as we have discussed in this guide. One of the best ways to design this new class of applications is through event-driven architecture (EDA) and domain-driven design (DDD), in particular events-first domain-driven design, which we will discuss next.

Events-First Domain-Driven Design

The term *events-first domain-driven design*, coined by software developer Russell Miles, refers to a set of design principles that focus on building scalable distributed systems. Rather than centering on *domain objects* (nouns), this approach emphasizes the *events* (verbs) within the domain.

Shifting the focus to events offers a clearer understanding of the domain's dataflow and communication patterns. This perspective helps capture the essence of the domain and naturally leads to scalable, event-driven architectures, which are well-suited for distributed systems at scale.

Focus on the Behavior of the System

Object-oriented programming (OOP) and DDD traditionally start by focusing on the *things*—the *nouns*—in the domain as a way of finding the domain objects. They then build up the design from there. The problem is that this approach has a major flaw: it forces us to focus on *structure* too early.

Instead, we should focus on the *events*—the *verbs* in the domain. This shift helps us understand how changes propagate, including communication patterns, workflows, and data ownership. By modeling the domain through events, we can better grasp how different components interact and who is responsible for what.

As Greg Young, the creator of CQRS (see "Untangle Reads and Writes" on page 54), says: "When you start modeling events, it forces you to think about the behavior of the system, as opposed to thinking about structure inside the system."

Focusing on events introduces a temporal dimension, making time a critical element in system design. Understanding the causal relationships between events is especially valuable in distributed systems, where managing time and change is crucial for reliable operation.

Use Commands for Intent and Events for Facts

Commands express an *intent to perform an action*, often resulting in side effects, such as changing internal state, triggering processes, or sending more commands.

Events represent *facts* about the domain and should be a key part of the ubiquitous language, helping to define the bounded context and establishing a clear boundary for the service. *Facts* represent things that have happened in the past and are immutable, or as defined by Merriam-Webster: "an actual occurrence; something that has actual existence":

Facts are immutable.
: They can't be changed or be retracted. We can't change the past, even if we sometimes wish that we could.

Knowledge is cumulative.
: This occurs either by receiving new facts or by deriving new facts from existing facts.

Existing knowledge can be invalidated.
: This is done by adding new facts to the system that refute existing facts. Facts are not deleted, only made irrelevant to current knowledge.

Thus, commands and events serve distinct roles in a system, and they differ in semantics, behavior, and purpose. Table 4-1 breaks it down.

Table 4-1. The differences between commands and events

Commands	Events
All about intent	Intentless
Directed (sender to receiver)	Anonymous (no address)
Clear single destination (retaining server address)	Just happens for others (0–*N*) to observe
Mimics a conversation (personal one-to-one direct communication)	Mimics broadcast ("speaker's corner"; does not care who is listening)
Distributed focus (moves across address spaces)	Local focus (no notion of communication, network, or addresses; events need to be wrapped in messages to travel address spaces)
Command and control	Autonomy

As we can see, in event-driven design, events are immutable facts published for anyone who is interested (can be zero or many), while commands express an intent to trigger actions or state changes in some other component. Events and commands serve different roles, with events representing what has happened and commands initiating future actions.

Understand Causality by Mining the Facts

You should carefully mine the facts to uncover causal relationships, as these are crucial for understanding the domain and the system itself. To model causality, a centralized approach is event logging (see "Log Events" on page 51), while decentralized methods utilize vector clocks or CRDTs (see "Accept Uncertainty" on page 17).

The technique of *event storming*[5] facilitates this process by gathering all stakeholders—domain experts and programmers—into a single

5 An in-depth discussion of event storming is beyond the scope of this guide, but a good starting point is Alberto Brandolini's book *Event Storming*.

room to brainstorm using sticky notes. They work together to define the domain language for events and commands, exploring the causal relationships and understanding the reactions.

The process involves three key steps:

1. *Explore the domain* by examining what happens in the system to identify events and their causal connections.
2. *Identify the triggers* for these events, which often result from the execution of commands, including user interactions, requests from other services, and inputs from external systems.
3. *Determine the endpoints* for commands, which are typically received by a component that decides whether to execute the side effects, potentially creating events that introduce new facts into the system.

This structured approach enables a clear understanding of the domain, revealing the flow of commands and events, and now directs our attention to the components (the entities, discussed in "Model Consistency Boundaries with Entities" on page 113), where these events culminate—our source of truth.

I find it helpful to design with *consistency boundaries*[6] in mind for services by following these steps:

1. Avoid starting with the service's behavior; instead, focus on the underlying data—what the facts are—and consider their coupling and dependencies.
2. Identify and model the integrity constraints that need to be upheld, gathering insights from domain experts and stakeholders throughout the process.
3. Start with zero guarantees on the smallest possible dataset, gradually incorporating the minimal level of guarantees needed to resolve your issue while keeping the dataset size small.
4. Use the single responsibility principle as a guiding concept.

6 Pat Helland's paper "Data on the Outside Versus Data on the Inside" talks about guidelines for designing consistency boundaries. It is essential reading for anyone building microservices-based systems.

The objective is to minimize the dataset that requires strong consistency. Once you have defined the essential dataset for the service, you can then address the behavior and protocols for interacting with other services, thereby establishing your unit of consistency.

Localize Mutable State and Publish Facts

> The assignment statement is the von Neumann bottleneck of programming languages and keeps us thinking in word-at-a-time terms in much the same way the computer's bottleneck does.
>
> —John Backus (Turing Award lecture, 1977)

After this extensive discussion of events and immutable facts, you may question whether mutable state has any role to play at all.

While mutable state—typically represented as variables—can be problematic, it's important to recognize its nuances. One issue is that the assignment statement, as highlighted by John Backus in his Turing Award lecture, is a destructive operation that overwrites previous data, effectively resetting time and history repeatedly.

The essence of the problem (as discussed in "Communicate Facts" on page 38) is that—as Rich Hickey, the inventor of the Clojure programming language, has discussed—most object-oriented computer languages (like Java, C++, and C#) treat the concepts of *value* and *identity* as the same thing, which means that an identity can't be allowed to evolve without changing the value it currently represents.

Functional languages such as Scala, Haskell, and OCaml, which employ pure functions with immutable data, provide a robust framework for reasoning about programs, allowing for stable values that remain unchanged during observation.

However, this does not imply that all mutable state is inherently bad. It just needs to be contained, meaning that you should use mutable states only for *local computations*, within the safe haven that the service instance represents, completely unobservable by the rest of the world. When you are done with the local processing and are ready to tell the world about your results, you then create an immutable fact representing the result and publish it to the world (as discussed in "Isolate Mutations" on page 40).

This approach allows others to rely on stable values for reasoning while still reaping the benefits of mutability, such as simplicity and algorithmic efficiency.

All these principles, patterns, and tools can be overwhelming to think about from first principles and put in practice. We need higher-level abstractions and guardrails to keep us on the right path. An event-based approach, for example, relying on entities and event sourcing (see "Log Events" on page 51), can naturally guide us toward adopting the best architecture and making the right trade-offs.

Model Consistency Boundaries with Entities

The *consistency boundary* (see "Isolate Mutations" on page 40) defines both a unit of consistency and a unit of failure (see "Embrace Failure" on page 22), ensuring that updates, relocations, and failures occur atomically. When transitioning from a monolith with a single database schema, it's essential to apply *denormalization techniques* to split the schema into multiple, manageable ones (see "Partition State" on page 37).

Each unit of consistency should be treated as an entity, and entities should reference one another by identity (see "Model with Actors" on page 66) rather than with direct references. This practice maintains isolation, reduces memory consumption by avoiding eager loading, and facilitates location transparency (see "Leverage Location Transparency" on page 49).

Entities that don't reference one another directly can be repartitioned and moved around in the cluster for almost infinite scalability,[7] as outlined by Pat Helland in his influential paper "Life Beyond Distributed Transactions".

Outside the entity's consistency boundary, eventual consistency is necessary. A good question to ask is: "Who is responsible for ensuring data consistency?" If the responsibility lies with the service executing the business logic, it should be confined to a single entity to guarantee strong consistency; otherwise, eventual consistency is acceptable.

7 You can find a good summary of the design principles in "7 Design Patterns for Almost-Infinite Scalability".

Suppose that we need to understand how an order management system works. After a successful event-storming session, we might end up with the following (drastically simplified) design:

- *Commands*: `CreateOrder`, `SubmitPayment`, `ReserveProducts`, `ShipProducts`
- *Events*: `OrderCreated`, `ProductsReserved`, `PaymentApproved`, `PaymentDeclined`, `ProductsShipped`
- *Entities*: `Orders`, `Payments`, `Inventory`

Figure 4-5 presents the flow of commands between a client and the services/entities (a dotted-line arrow indicates that the command or event was sent asynchronously).

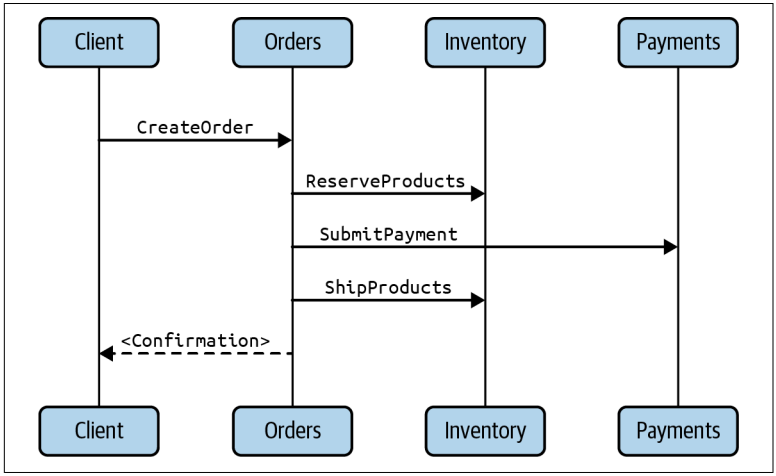


Figure 4-5. The flow of commands in an example order management use case

If we add the events to the picture, it looks something like the flow of commands shown in Figure 4-6.

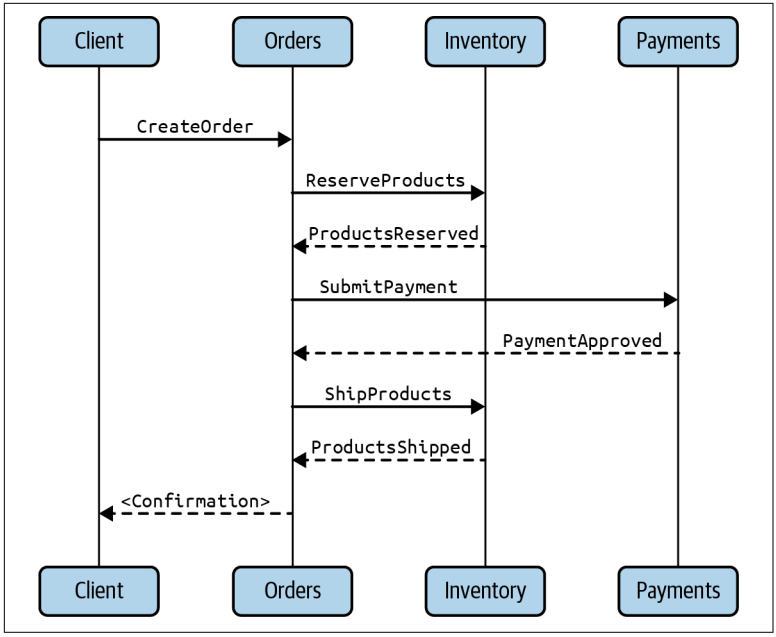


Figure 4-6. The flow of commands and events in an example order management use case

Let's now zoom in on the payments service and how it interacts with the orders service. Figure 4-7 (that you might remember from "Log Events" on page 51) illustrates the flow of the commands and events in the interaction between these services. It also shows how event sourcing and CQRS come into play (see "Log Events" on page 51 and "Untangle Reads and Writes" on page 54).

The process begins with a command (`SubmitPayment`) sent to the payments service from an external "user" of the service, namely orders. This command crosses the boundary of the bounded context for the service and is received by the processing layer, where it undergoes validation and translation before the business logic is executed. Following this, a new command (`ApprovePayment`) is created, representing the intent to change the service's state, and sent to the service's entity. The entity then processes the command, generating an event (`PaymentApproved`) that signifies the state change and storing it in its event log. Once the event is successfully saved, it is pushed to the event stream for public consumption, where it is forwarded to its subscribers, which include the orders service and the payment query database (the CQRS "read side").

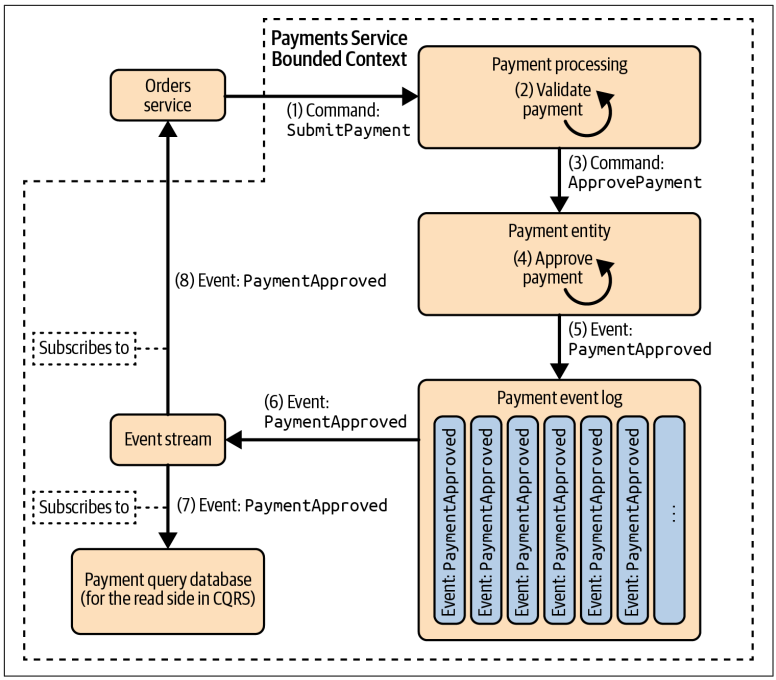


Figure 4-7. A simple ordering system using event sourcing and CQRS

This design enhances scalability, resilience, and consistency, allowing services to resume processing after failures or replay portions of the event stream if needed.

Allow for Protocol Evolution

A *protocol* defines the rules and etiquette for exchanging messages between components. It outlines the relationships between participants, the current state of the protocol, and the set of allowed messages at any time, essentially specifying which messages one participant can send to another at each step.

Protocols can be categorized by the type of interaction, such as request–reply, repeated request–reply (like HTTP), publish–subscribe, and streaming (both push and pull). Unlike a local programming interface, a protocol is more flexible, supporting multiple participants and tracking the ongoing state of the exchange, rather than just one-time interactions. Note that a protocol only defines which messages can be sent, while encoding/decoding (codecs) and

transport are implementation details independent of the protocol itself.

Individual microservices remain independent and decoupled only when they can evolve autonomously, necessitating that their protocols (e.g., defined through HTTP APIs, RPC, events published or subscribed to, etc.) be resilient to and permissive of change. This includes handling events and commands, persistently stored data, and the exchange of ephemeral information such as session states, authentication credentials, and cached data. Ensuring interoperability between different versions is essential for the effective long-term management of complex service ecosystems.

Postel's law,[8] also known as the *robustness principle*, states that you should "be conservative in what you do, be liberal in what you accept from others," making it a valuable guideline for API design and evolution in collaborative services.[9]

Challenges in managing microservices include the versioning of protocols and data—specifically events and commands—and addressing how to handle upgrades and downgrades effectively. This complex problem encompasses several key tasks:

- Selecting extensible codecs for serialization
- Ensuring that incoming commands are valid
- Maintaining a translation layer for protocols and data, which may require upgrading or downgrading events or commands to align with the current version
- Versioning of the service itself[10]

To effectively handle these tasks, implementing an anti-corruption layer is advisable. This layer can be integrated into the service or managed through an API gateway, serving to protect the bounded context from changes in other contexts while allowing both to evolve independently.

8 Originally stated by Jon Postel in RFC 761 on TCP in 1980.

9 It has, among other things, influenced the tolerant reader pattern.

10 There is a semantic difference between a service that is truly new and a new version of an existing service.

Congratulations on making it this far. Hopefully now you have a good understanding of the principles and patterns that are the foundation for solid distributed-system architecture and design, as well as how to leverage microservices and event-driven design to put them into practice building cloud and edge applications.

Conclusion

We have covered a lot of ground in this guide, yet have just scratched the surface of some of the topics. I hope what you have read has inspired you to learn more and to roll up your sleeves and try these ideas out in practice.

Learning from past failures[11] and successes[12] in distributed systems and collaborative services-based architectures is paramount. Thanks to books and papers, we don't need to start from scratch but can learn from other people's successes, failures, mistakes, and experiences.

There are a lot of references (in hyperlinks and footnotes) throughout this guide, and I very much encourage you to read them all.

11 The failures of SOA, CORBA, EJB, and synchronous RPC are well worth studying and understanding.

12 Successful platforms with tons of great design ideas and architectural patterns have so much to teach us—for example, Tandem Computer's NonStop platform, the Erlang platform, and the BitTorrent protocol.

About the Author

Jonas Bonér is a distributed systems innovator and OSS community leader. He founded Akka (née Lightbend) in 2009 and authored the *Reactive Manifesto*, *Reactive Principles*, and numerous distributed systems design books that set the foundation for how event-driven systems are built today. He has created Akka, Cloudstate, Lagom, Kalix, and the AspectWerkz Aspect-Oriented Programming (AOP) framework (now merged with the Eclipse AspectJ project). Jonas is an amateur jazz musician, a passionate skier, and holds a Bachelor of Science from Mid Sweden University.