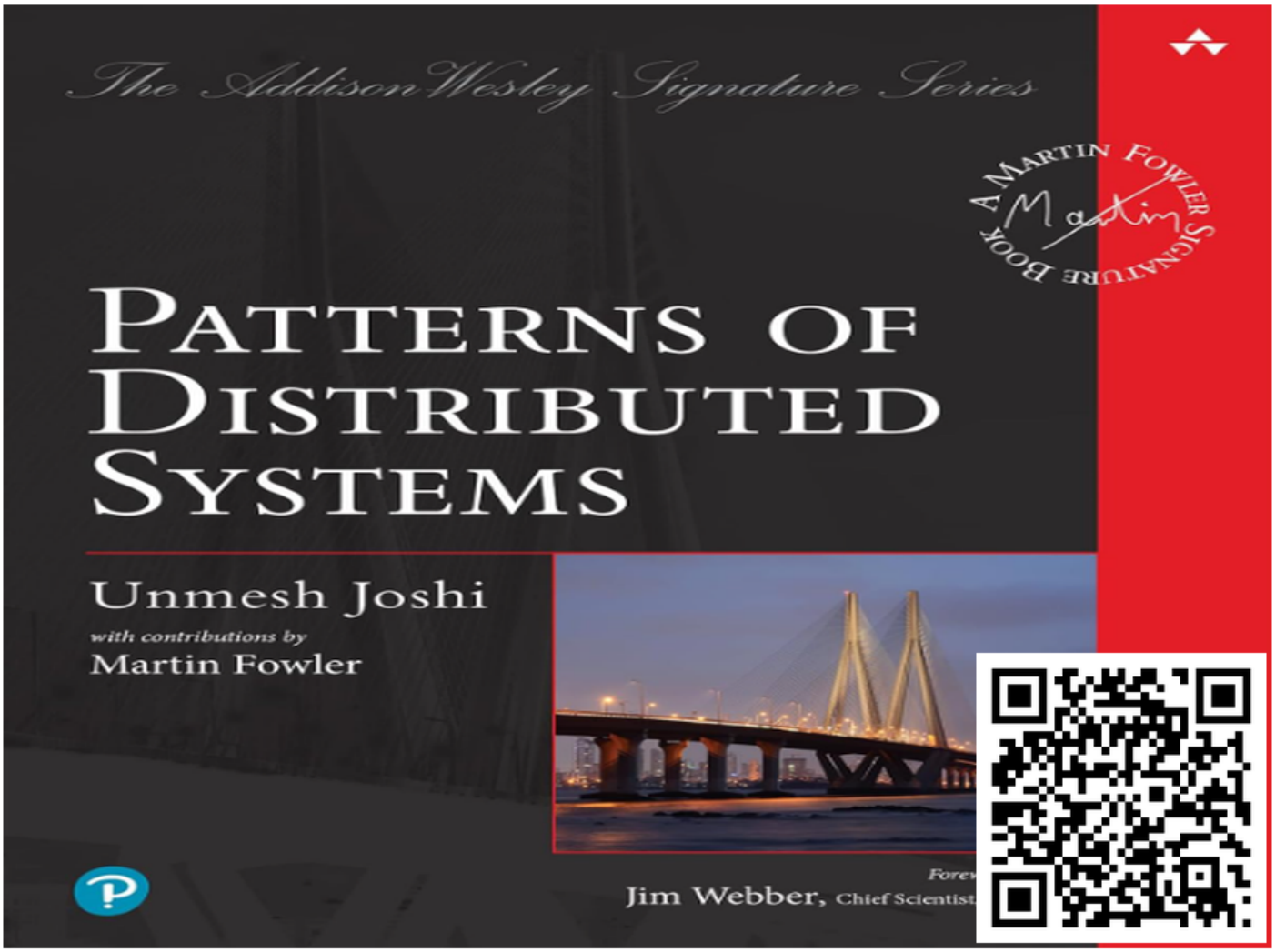

The Addison-Wesley Signature Series

A Martin Fowler Signature Book

Patterns of Distributed Systems

Unmesh Joshi

with contributions by

Martin Fowler

Foreword by

Jim Webber, Chief Scientist, Neo4j

Patterns of Distributed Systems

Pearson Addison-Wesley Signature Series

Visit **informit.com/awss** for a complete list of available publications.

The **Pearson Addison-Wesley Signature Series** provides readers with practical and authoritative information on the latest trends in modern technology for computer professionals. The series is based on one simple premise: great books come from great authors.

Books in the Martin Fowler Signature Series are personally chosen by Fowler, and his signature ensures that he has worked closely with authors to define topic coverage, book scope, critical content, and overall uniqueness. The expert signatures also symbolize a promise to our readers: you are reading a future classic.

Patterns of Distributed Systems

Unmesh Joshi

Addison-Wesley

Hoboken, New Jersey

Cover image: Joe Ravi/Shutterstock
Additional image credits appear on page 427, which constitutes a continuation of this copyright page.

Visit us on the Web: informit.com/aw

Library of Congress Control Number: 2023944564

ISBN-13: 978-0-13-822198-0
ISBN-10: 0-13-822198-7

$PrintCode

Pearson's Commitment to Diversity, Equity, and Inclusion

Pearson is dedicated to creating bias-free content that reflects the diversity of all learners. We embrace the many dimensions of diversity, including but not limited to race, ethnicity, gender, socioeconomic status, ability, age, sexual orientation, and religious or political beliefs.

Education is a powerful force for equity and change in our world. It has the potential to deliver opportunities that improve lives and enable economic mobility. As we work with authors to create content for every product and service, we acknowledge our responsibility to demonstrate inclusivity and incorporate diverse scholarship so that everyone can achieve their potential through learning. As the world's leading learning company, we have a duty to help drive change and live up to our purpose to help more people create a better life for themselves and to create a better world.

Our ambition is to purposefully contribute to a world where:

- Everyone has an equitable and lifelong opportunity to succeed through learning.
- Our educational products and services are inclusive and represent the rich diversity of learners.
- Our educational content accurately reflects the histories and experiences of the learners we serve.
- Our educational content prompts deeper discussions with learners and motivates them to expand their own learning (and worldview).

While we work hard to present unbiased content, we want to hear from you about any concerns or needs with this Pearson product so that we can investigate and address them.

- Please contact us with concerns about any potential bias at `https://www.pearson.com/report-bias.html`.

This page intentionally left blank

Dedicated to the loving memory of
my father.

This page intentionally left blank

Contents

Foreword

Engineers are often attracted to distributed computing, which promises not only benefits like scalability and fault tolerance but also the prestige of creating clever, talk-worthy computer systems. But the reality is that distributed systems are hard. There are myriads of edge cases, all with subtle interactions and high-dimensional nuance. Every move you make as a systems designer has *n*-th degree side effects which aren't obvious. You're Sideshow Bob, surrounded by lawn rakes, and every step you take results in a rake in the face—until you've left the field or expended all the rakes. (Oh, and even when you've left the field, there's still a rake or two waiting to be trodden on.)

So how do we avoid, or at least minimize, these pitfalls? The traditional approach has been to accept that distributed systems theory and practice are both hard, and to work your way through textbooks and academic papers with confusing or playful titles, studying numerous proofs so that you can carve out small areas of relative safety and expertise within which to build your system. There's a lot of value in that approach for those that can stay the course. Systems professionals who have grown up that way seem to have a knack for spotting trouble far down the line, and possess a good deal of technical background for reasoning about how to solve problems—or at least minimize their likelihood or impact.

However, in other areas of software engineering, this kind of educational hazing is not so commonplace. Instead of being thrown in at the deep end, we use abstractions to help us gradually learn at greater levels of detail, from higher to lower levels of abstraction, which often maps neatly onto the way software is designed and built. Abstractions allow us to reason about behaviors without getting bogged down in implementation complexity. In a distributed system where complexity is high, some abstractions can be very useful.

In general software engineering, design patterns are a common abstraction. A design pattern is a standardized solution to a recurrent problem in software design. Patterns provide a language that practitioners use to reason about and discuss problems in a well-understood manner. For example, when someone asks, "How does this work?" you may hear something like, "It's just a visitor." Such exchanges,

based on a shared understanding of named patterns that solve common problems, are short and information-rich.

The notion of taking something complex and abstracting it into a pattern is both important and fundamental to this book. It applies the pattern approach to the essential building blocks of modern distributed systems, naming the components and describing their behaviors and how they interact. In doing so, it equips you with a pattern language that, within reason, lets you treat a distributed system as a set of composable Lego blocks.

Now, you can talk about "a system that depends on a replicated log with quorum commits" without getting bogged down in the specific details of the data structures and consensus algorithms. Perhaps more importantly, it minimizes the risk of talking past one another because in distributed systems, textbook terms—such as "consistency"—often have several meanings depending on context.

The effect is liberating for practitioners who now have an expressive common vocabulary to expedite and standardize communication. But it's also liberating for learners who can take a structured, breadth-first tour of distributed systems fundamentals, tackling a pattern at a time and observing how those patterns interact or depend on one another. You can also, where needed, go deep into the implementation—this book does not shy away from implementation details either.

My hope is that the patterns in this book will help you teach, learn, and communicate more effectively about distributed systems. It will certainly help you avoid some of the lawn rakes.

—Jim Webber, Chief Scientist, Neo4j

Preface

Why This Book

In 2017, I was involved in developing a software system for a large optical telescope called Thirty Meter Telescope (TMT). We needed to build a core framework and services to be used by various subsystems. The subsystem components had to discover each other and detect component failures. There was also a requirement to store metadata about these components. The service responsible for storing this information had to be fault-tolerant. We couldn't use off-the-shelf products and frameworks due to the unique nature of the telescope ecosystem. We had to build it all from scratch—to create a core framework and services that different subsystems of the software could use. In essence, we had to build a distributed system.

I had designed and architected enterprise systems using products such as Kafka, Cassandra, and MongoDB or cloud services from providers like AWS and GCP. All these products and services are distributed and solve similar problems. For the TMT system, we had to build a solution ourselves. To compare and validate our implementation choices with these proven products, we needed a deeper understanding of the inner workings of some of these products. We had to figure out how all these cloud services and products are built and why they are built that way. Their own documentation often proved too product-specific for that.

Information about how distributed systems are built is scattered across various research papers and doctoral theses. However, these academic sources have their limitations too. They tend to focus on specific aspects, often making only passing references to related topics. For instance, consider a well-written thesis, *Consensus: Bridging Theory and Practice* [Ongaro2014]. It thoroughly explains how to implement the Raft consensus algorithm. But you won't know how Raft is used by products like etcd for tracking group membership and related metadata for other products, such as Kubernetes. Leslie Lamport's famous paper "Time, Clocks, and the Ordering of Events in a Distributed System" [Lamport1978] talks about how to use

logical clocks—but you won't know how products like MongoDB use them as a version for the data they store.

I believe that writing code is the best way to test your understanding. Martin Fowler often says, "Code is like the mathematics of our profession. It's where we have to remove the ambiguity." So, to get a deeper understanding of the building blocks of distributed systems, I decided to build miniature versions of these products myself. I started by building a toy version of Kafka. Once I had a reasonable version, I used it to discuss some of the concepts of distributed systems. That worked well. To verify that explaining concepts through code works effectively, I conducted a few workshops within my company, Thoughtworks. Those turned out to be very useful. So I extended this to products like Cassandra, Kubernetes, Akka, Hazelcast, MongoDB, YugabyteDB, CockroachDB, TiKV, and Docker Swarm. I extracted code snippets to understand the building blocks of these products. Not surprisingly, there were a lot of similarities in these building blocks. I happened to discuss this with Martin Fowler a few years back, and he suggested writing about these as patterns. This book is the outcome of my work with Martin to document these common building blocks of distributed system implementations as patterns.

Who This Book Is For

Software architects and developers today face a plethora of choices when it comes to selecting products and cloud services that are distributed by design. These products and services claim to make certain implementation choices. Understanding these choices intuitively can be challenging. Just reading through the documentation is not enough. Consider sentences like "AWS MemoryDB ensures durability with a replicated transactional log" or "Apache Kafka operates independently from ZooKeeper" or "Google Spanner provides external consistency with accurate timing maintained by TrueTime." How do you interpret these?

To get better insights, professionals rely on certifications from product providers. But most certifications are very product-specific. They focus only on the surface features of the product but not the underlying technical principles. Professional developers need to have an intuitive understanding of technical details that are specific enough to be expressed at the source-code level but generic enough to apply to a wide range of situations. Patterns help there. Patterns in this book will enable working professionals to have a good idea of what's happening under the hood of various products and services and thus make informed and effective choices.

I expect most readers of this book to be in this group. In addition to those who work with existing distributed systems, however, there is another group of readers

who must build their own distributed systems. I hope the patterns in this book will give that other group a head start. There are numerous references to design alternatives used by various products, which might be useful to these readers.

A Note on Examples

I have provided code examples for most of the patterns. The code examples are based on my own miniature implementations of the various products I studied while working through these patterns. My choice of language is based on what I think most readers are likely to be able to read and understand. Java is a good choice here. The code examples use a minimum of Java language features—mostly methods and classes, which are available in most programming languages. Readers familiar with other programming languages should be able to easily understand these code examples. This book, however, is not intended to be specific for any particular software platform. Once you understand the code examples, you will find similarities in code bases in C++, Rust, Go, Scala, or Zig. My hope is that, once you are familiar with the code examples and the patterns, you will find it easier to navigate the source code of various open-source products.

How to Read This Book

The book has six numbered parts that are divided into two main conceptual sections.

First, a number of narrative chapters cover the essential topics in distributed systems design. These chapters (in Part I) present challenges in distributed system design along with their solutions. However, they don't go into much detail on these solutions.

Detailed solutions, structured as patterns, are provided in the second section of the book (Parts II to VI). The patterns fall into four main categories: replication, partitioning, cluster management, and network communication. Each of these is a key building block of a distributed system.

Consider these patterns as references; there's no need to read them cover to cover. You may read the narrative chapters for an overview of the book's scope, and then explore the patterns based on your interests and requirements.

For additional reference materials, visit `https://martinfowler.com/articles/patterns-of-distributed-systems`.

I hope these patterns will assist fellow software professionals in making informed decisions in their daily work.

Register your copy of *Patterns of Distributed Systems* on the InformIT site for convenient access to updates and/or corrections as they become available. To start the registration process, go to `informit.com/register` and log in or create an account. Enter the product ISBN (9780138221980) and click Submit. Look on the Registered Products tab for an Access Bonus Content link next to this product, and follow that link to access any available bonus materials. If you would like to be notified of exclusive offers on new editions and updates, please check the box to receive email from us.

Acknowledgments

First and foremost, the book was only possible because of encouragement from Martin Fowler. He guided me to think in terms of patterns. He also helped me come up with good examples and contributed to the chapters that were very tricky to write.

I want to thank the Thirty Meter Telescope (TMT) team. Working with that team was the trigger for much of this work. I had good conversations about many of these patterns with Mushtaq Ahmed who was leading the TMT project.

Sarthak Makhija validated a lot of these patterns while he worked on building a distributed key-value store.

I have been publishing these patterns periodically on `martinfowler.com`. While working on these patterns, I sent drafts of new material to the Thoughtworks developer mailing list and asked for feedback. I want to thank the following people for posting their feedback on the mailing list: Rebecca Parsons, Dave Elliman, Samir Seth, Prasanna Pendse, Santosh Mahale, James Lewis, Chris Ford, Kumar Sankara Iyer, Evan Bottcher, Ian Cartwright, and Priyanka Kotwal. Jojo Swords, Gareth Morgan, and Richard Gall from Thoughtworks helped with copyediting the earlier versions published on `martinfowler.com`.

While working on the patterns, I interacted with many people. Professor Indranil Gupta provided feedback on the Gossip Dissemination pattern. Dahlia Malkhi helped with questions about Google Spanner. Mikhail Bautin, Karthik Ranganathan, and Piyush Jain from the Yugabyte team answered all my questions about some of implementation details in YugabyteDB. The CockroachDB team was very responsive in answering questions about their design choices. Bela Ban, Patrik Nordwall, and Lalith Suresh provided good feedback on the Emergent Leader pattern.

Salim Virji and Jim Webber went through the early manuscript and provided some nice feedback. Richard Sites provided some nice suggestions on the first chapter. I want to extend my heartfelt thanks to Jim Webber for contributing the foreword to this book.

One of the great things about being an employee at Thoughtworks is that they allowed me to spend considerable time on this book. Thanks to the Engineering

for Research (E4R) group of Thoughtworks for their support. I want to also thank Sameer Soman, MD, Thoughtworks India, who always encouraged me.

At Pearson, Greg Doench is my acquisition editor, navigating many issues in getting a book to publication. I was glad to work with Julie Nahil as my production editor. It was great to work with Dmitry Kirsanov for copyediting and Alina Kirsanova for composition and indexing.

My family has been a source of constant support. My mother was always very hopeful about the book. My wife, Ashwini, is an excellent software developer herself; she and I had insightful discussions and she provided valuable reviews of early drafts. My daughter, Rujuta, and son, Advait, were sources of my motivation.

About the Author

Unmesh Joshi is a Principal Consultant at Thoughtworks with 24 years of industry experience. As an ardent enthusiast of software architecture, he firmly believes that today's tech landscape requires a profound understanding of distributed systems principles. For the last three years he has been publishing patterns of distributed systems on martinfowler.com. He has also conducted various training sessions around this topic. You can find him on X (formerly Twitter): @unmeshjoshi.

This page intentionally left blank

Part I

Narratives

This page intentionally left blank

Chapter 1

The Promise and Perils of Distributed Systems

The Limits of a Single Server

In this book, we will discuss distributed systems. But what exactly do we mean when we say "distributed systems"? And why is distribution necessary? Let's start from the basics.

In today's digital world, the majority of our activities rely on networked services. Whether it's ordering food or managing finances, these services run on servers located somewhere. When using cloud services like AWS, GCP, or Azure, these servers are managed by the respective cloud providers. They store data, process user requests, and perform computations using the CPU, memory, network, and disks. These four fundamental physical resources are essential for any computation.

Consider a typical retail application functioning as a networked service, where users can perform actions such as adding items to their shopping cart, making purchases, viewing orders, and querying past orders. The capacity of a single server to handle user requests is ultimately determined by the limitations of four key resources: network bandwidth, disks, CPU, and memory.

The network bandwidth sets the maximum data transfer capacity over the network at any given time. For example, with a network bandwidth of 1Gbps (125MB/s) and 1KB records being written or read, the network can support a maximum of 125,000 requests per second. However, if the record size increases to 5KB, the number of requests that can be passed over the network decreases to only 25,000.

Disk performance depends on several factors, including the type of read or write operations and how well disk caches are used. Mechanical disks are also affected by hardware features such as rotational speed and seek time. Sequential operations usually have better performance than random ones. Moreover, the performance is influenced by concurrent read/write operations and software-based transactional processes. These factors can significantly affect the overall throughput and latency on a single server.

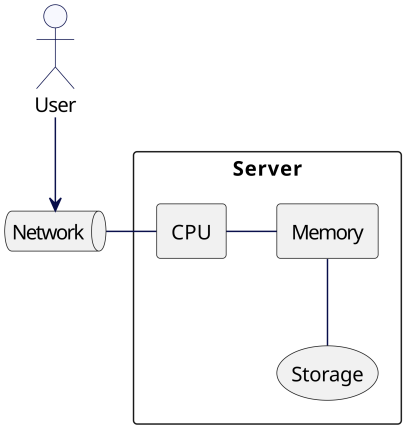

Figure 1.1 *Resources of computation*

Likewise, when the CPU or memory limit is reached, requests must wait for their turn to be processed. When these physical limits are pushed to their capacity, this results in queuing. As more requests pile up, waiting times increase, negatively impacting the server's ability to efficiently handle user requests.

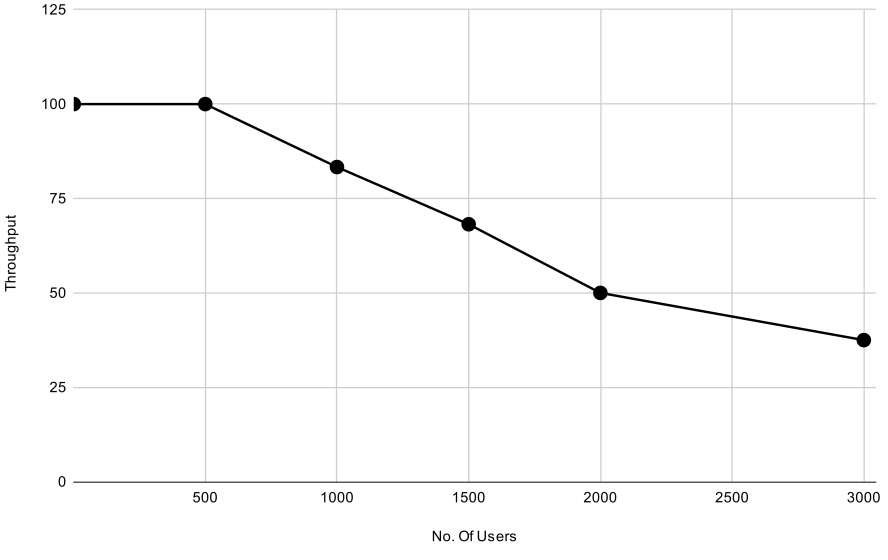

Figure 1.2 *Drop in throughput with increase in requests*

The impact of reaching the limits of these resources becomes evident in the overall throughput of the system, as illustrated in Figure 1.2.

This poses a problem for end users. As the system is expected to accommodate an increasing user base, its performance actually degrades.

To ensure requests are served effectively, you have to divide and process them on multiple servers. This enables the utilization of separate CPUs, networks, memory, and disks to handle user requests. In our example, the workload should be divided so that each server handles approximately five hundred requests.

Separate Business Logic and Data Layer

A common approach is to separate an architecture into two parts. The first part is the stateless component responsible for exposing functionality to end users. This can take the form of a web application or, more commonly, a web API that serves user-facing applications. The second part is the stateful component, which is managed by a database (Figure 1.3).

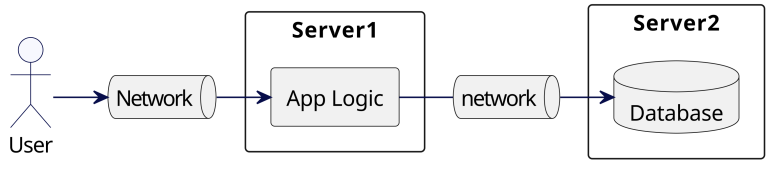

Figure 1.3 *Separating compute and data*

This way, most of the application logic executes on the separate server utilizing a separate network, CPU, memory, and disk. This architecture works particularly well if most users can be served from caches put at different layers in the architecture. It makes sure that only a portion of all requests need to reach the database layer.

As the number of user requests increases, more servers can be added to handle the stateless business logic. This scalability allows the system to accommodate a growing user base and ensures that requests can be processed efficiently. In the event of a server failure, a new server can be introduced to take over the workload and continue serving user requests seamlessly (Figure 1.4).

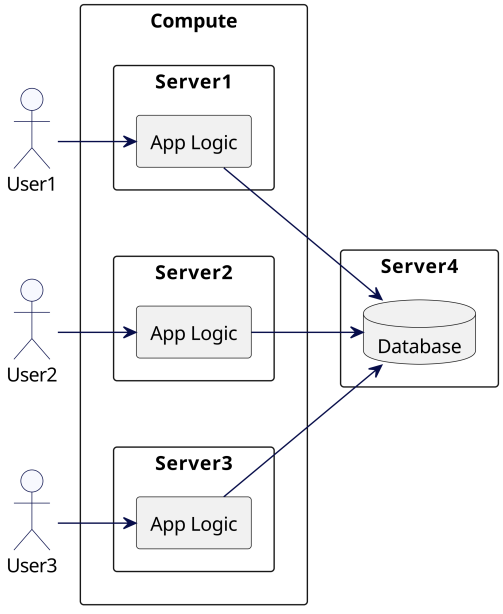

Figure 1.4 *Scaling compute with multiple servers*

This approach is effective for many applications. However, there comes a point when the amount of data stored in stateful databases grows to hundreds of terabytes or even petabytes, or the number of requests to the database layer increases significantly. As a result, the simplistic architecture described above runs into limitations stemming from the physical constraints of the four fundamental resources on the server responsible for managing the data.

Partitioning Data

When a software system runs into hardware's physical limits, the best approach to ensure proper request processing is to divide the data and process it on multiple servers (Figure 1.5). This enables the utilization of separate CPUs, networks, memory, and disks to handle requests on smaller data portions.

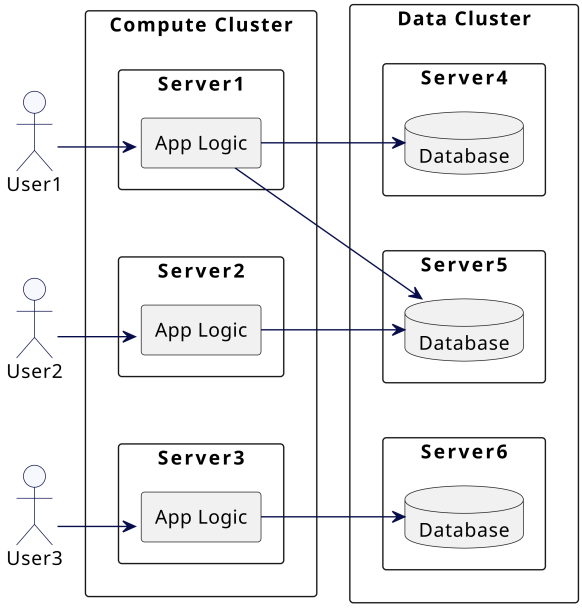

Figure 1.5 *Scaling data by distributing on multiple servers*

A Look at Failures

When we utilize multiple machines with their own disk drives, network interconnects, processors, and memory units, the likelihood of failures becomes a significant concern. Consider the hard disk failure probability. If a disk has a failure rate of once in 1000 days, the probability of it failing on any given day is 1/1000, which may not be a major concern on its own. However, if we have 1000 disks, the probability of at least one disk failing on a given day becomes 1. If the partitioned data is being served from the disk that fails, it will become unavailable until the disk is recovered.

To gain insights into the types of failures that can occur look at the failure statistics from Jeff Dean's 2009 talk [Dean2009] on Google's data centers as shown in Table 1.1. Although these numbers are from 2009, they still provide a valuable representation of failure patterns.

Table 1.1 *Failure Events per Year for a Cluster in a Data Center from Jeff Dean's 2009 Talk [Dean2009]*

Event	Details
Overheating	Power down most machines in < 5 min (~1–2 days to recover)
PDU Failure	~500–1000 machines suddenly disappear (~6 hours to come back)
Rack Move	Plenty of warning, ~500–1000 machines powered down (~6 hours)
Network Rewiring	Rolling ~5% of machines down over 2-day span
Rack Failures	40–80 machines instantly disappear (1–6 hours to get back)
Racks Go Wonky	40–80 machines see 50% packet loss
Network Maintenances	4 might cause ~30-minute random connectivity losses
Router Reloads	Takes out DNS and external VIPs for a couple minutes
Router Failures	Have to immediately pull traffic for an hour
Minor DNS Blips	Dozens of 30-second blips for DNS
Individual Machine Failures	1000 individual machine failures
Hard Drive Failures	Thousands of hard drive failures

When distributing stateless compute across multiple servers, failures can be managed relatively easily. If a server responsible for handling user requests fails, the requests can be redirected to another server, or a new server can be added to take over the workload. Since stateless compute does not rely on specific data stored on a server, any server can begin serving requests from any user without the need to load specific data beforehand.

Failures become particularly challenging when dealing with data. Creating a separate instance on a random server is not as straightforward. It requires careful consideration to ensure that the servers start in the correct state and coordinate with other nodes to avoid serving incorrect or stale data. This book mainly focuses on systems that face these types of challenges.

To ensure that the system remains functional even if certain components are experiencing failures, simply distributing data across cluster nodes is often insufficient. It is crucial to effectively mask the failures.

Replication: Masking Failures

Replication plays a crucial role in masking failures and ensuring service availability. If data is replicated on multiple machines, even in the event of failures, clients can connect to a server that holds a copy of the data.

However, doing this is not as simple as it sounds. The responsibility for masking failures falls on the software that handles user requests. The software must be able to detect failures and ensure that any inconsistencies are not visible to the users. Understanding the types of errors that a software system experiences is vital for successfully masking these failures.

Let's look at some of the common problems that software systems experience and need to mask from the users of the system.

Process Crash

Software processes can crash unexpectedly due to various reasons. It could be a result of hardware failures or unhandled exceptions in the code. In containerized or cloud environments, monitoring software can automatically restart a process it recognizes as faulty. However, if a user has stored data on the server and received a successful response, it becomes crucial for the software to ensure that the data remains available after the process restarts. Measures need to be in place to handle process crashes and ensure data integrity and availability.

Network Delay

The TCP/IP network protocol operates asynchronously, meaning it does not provide a guaranteed upper bound on message delivery delay. This poses a challenge for software processes that communicate over TCP/IP. They must determine how long to wait for responses from other processes. If a response is not received within the designated time, they need to decide whether to retry or consider the other process as failed. This decision-making becomes crucial for maintaining the reliability and efficiency of communication between processes.

Process Pause

During the execution of a process, it can pause at any given moment. In garbage-collected languages like Java, execution can be interrupted by garbage collection pauses. In extreme cases, these pauses can last tens of seconds. As a result, other processes need to determine whether the paused process has failed. The situation becomes more complex when the paused process resumes and begins sending messages to other processes. The other processes then face a dilemma: Should they ignore the messages or process them, especially if they had previously

marked the paused process as failed? Finding the right course of action in these circumstances is a challenging problem.

Unsynchronized Clocks

The clocks in the servers typically utilize quartz crystals. However, the oscillation frequency of a quartz crystal can be influenced by factors like temperature changes or vibrations. This can cause the clocks on different servers to have different times. Servers typically require a service such as NTP[1] that continuously synchronizes their clocks with time sources over the network. However, network faults can disrupt this service, leading to unsynchronized clocks on servers.[2] As a result, when processes need to order messages or determine the sequence of saved data, they cannot rely on the system timestamps because clock timings across servers can be inconsistent.

Defining the Term "Distributed Systems"

We will explore the common solutions to address the challenges posed by these failures. However, before we delve into that, let's establish a definition for distributed systems based on our observations thus far.

A distributed system is a software architecture that consists of multiple interconnected nodes or servers working together to achieve a common goal. These nodes communicate with each other over a network and coordinate their actions to provide a unified and scalable computing environment.

In a distributed system, the workload is distributed across multiple servers, allowing for parallel processing and improved performance. The system is designed to handle large amounts of data and accommodate a high number of concurrent users. Most importantly, it offers fault tolerance and resilience by replicating data and services across multiple nodes, ensuring that the system remains operational even in the presence of failures or network disruptions.

The Patterns Approach

Professionals seeking practical advice need an intuitive understanding of these systems that goes beyond theory. They need detailed and specific explanations

1. Network Time Protocol.
2. Even Google's TrueTime clock machinery built using GPS clocks has clock skew. However, that clock skew has a guaranteed upper bound.

that help comprehend real code while remaining applicable to a wide range of systems. The Patterns approach is an excellent tool to fulfill these requirements.

The concept of patterns was initially introduced by architect Christopher Alexander in his book *A Pattern Language* [Alexander1977]. This approach gained popularity in the software industry, thanks to the influential book widely known as the *Gang Of Four* [Gamma1994] book.

Patterns, as a methodology, describe particular problems encountered in software systems, along with concrete solution structures that can be demonstrated by real code. One of the key strengths of patterns lies in their descriptive names and the specific code-level details they provide.

A pattern, by definition, is a "recurring solution" to a problem within a specific context. Therefore, something is only referred to as a pattern if it is observed repeatedly in multiple implementations. Generally, *The Rule of Three*[3] is followed—a pattern should be observed in at least three systems before it can be recognized as a pattern.

The patterns approach, employed in this book, is rooted in the study of actual codebases from various open source projects, such as Apache Kafka,[4] Apache Cassandra,[5] MongoDB,[6] Apache Pulsar,[7] etcd,[8] Apache ZooKeeper,[9] CockroachDB,[10] YugabyteDB,[11] Akka,[12] JGroups,[13] and others. These patterns are grounded in practical examples and can be applied to different software systems. By exploring the insights gained from these codebases, readers can learn to understand and apply these patterns to solve common software challenges.

Another important aspect of patterns is that they are not used in isolation but rather in conjunction with other patterns. Understanding how the patterns interlink makes it much easier to grasp the overall architecture of the system.

The next chapter takes a tour of most of the patterns and shows how they link together.

3. https://wiki.c2.com/?RuleOfThree
4. https://kafka.apache.org
5. https://cassandra.apache.org
6. https://www.mongodb.com
7. https://pulsar.apache.org
8. https://etcd.io
9. https://zookeeper.apache.org
10. https://www.cockroachlabs.com
11. https://www.yugabyte.com
12. https://akka.io
13. http://www.jgroups.org

This page intentionally left blank

Chapter 2

Overview of the Patterns

by Unmesh Joshi and Martin Fowler

As discussed in the last chapter, distributing data means at least one of two things: partitioning and replication. To start our journey through the patterns in this book, we'll focus on replication first.

Imagine a very minimal data record that captures how many widgets we have in four locations (Figure 2.1).

boston	50
philadelphia	38
london	20
pune	75

Figure 2.1 *An example data record*

We replicate it on three nodes: Jupiter, Saturn, and Neptune (Figure 2.2).

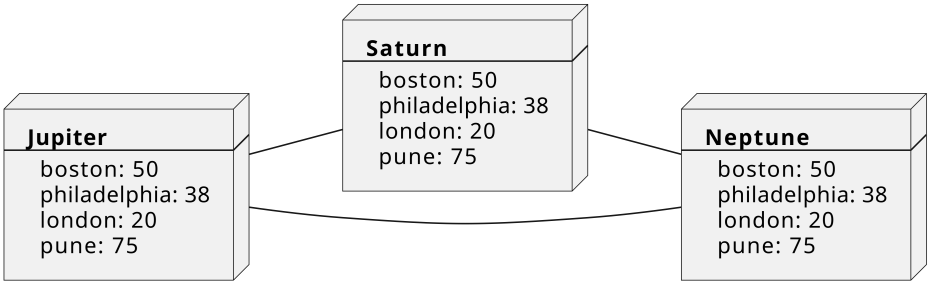

Figure 2.2 *Replicated data record*

Keeping Data Resilient on a Single Server

The first area of potential inconsistency appears with no distribution at all. Consider a case where the data for Boston, London, and Pune are held on different files. In this case, performing a transfer of 40 widgets means changing `bos.json` to reduce its count to 10 and changing `pnq.json` to increase its count to 115. But what happens if Neptune crashes after changing Boston's file but before updating Pune's? In that case we would have inconsistent data, destroying 40 widgets (Figure 2.3).

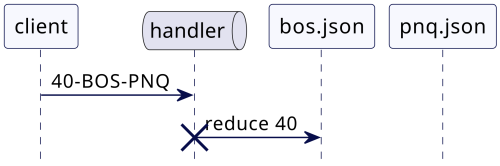

Figure 2.3 *Node crash causes inconsistency*

An effective solution to this is *Write-Ahead Log* (Figure 2.4). With this, the message handler first writes all the information about the required update to a

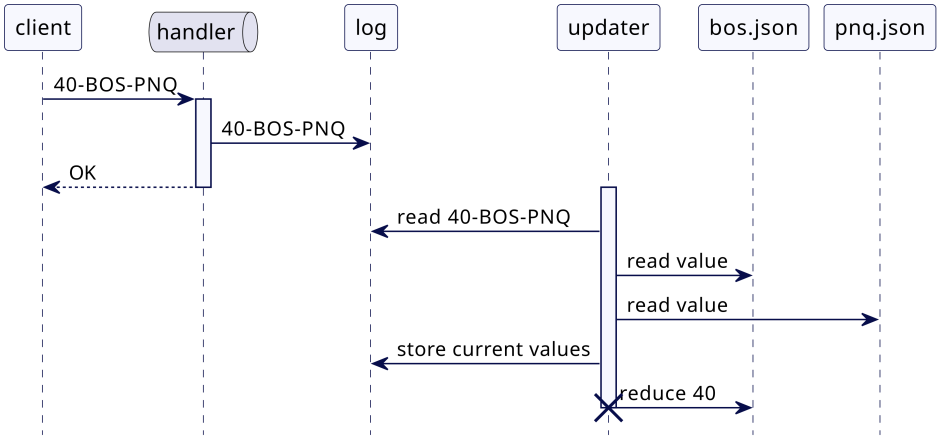

Figure 2.4 *Using WAL*

log file. This is a single write, so is simple to ensure it's done atomically. Once the write is done, the handler can acknowledge to its caller that it has handled the request. Then the handler, or other component, can read the log entry and carry out the updates to the underlying files.

Should Neptune crash after updating Boston, the log should contain enough information for Neptune, when it restarts, to figure out what happened and restore the data to a consistent state, as shown in Figure 2.5. (In this case it would store the previous values in the log before any updates are made to the data file.)

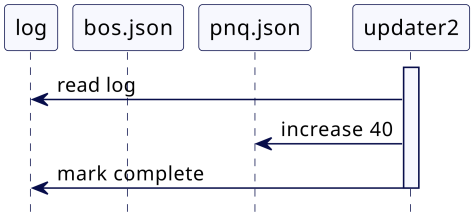

Figure 2.5 *Recovery using WAL*

The log gives us resilience because, for a known prior state, the linear sequence of changes determines the state after the log is executed. This property is important for resilience in a single node scenario but, as we'll see, it's also very valuable for replication. If multiple nodes start at the same state, and they all play the same log entries, we know they will end up at the same state too.

Databases use a Write-Ahead Log, as discussed in the above example, to implement transactions.

Competing Updates

Suppose two different users, Alice and Bob, are connecting to two different cluster nodes to execute their requests. Alice wants to move 30 widgets from Boston to London, while Bob wants to move 40 widgets from Boston to Pune (Figure 2.6).

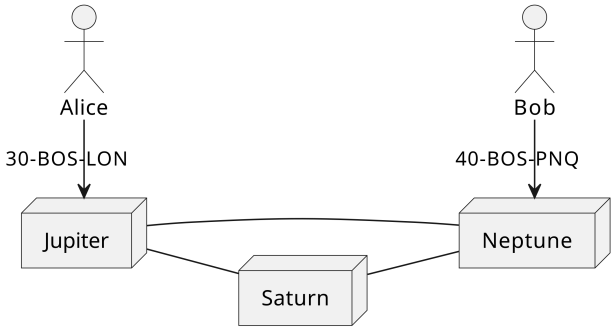

Figure 2.6 *Competing updates*

How should the cluster resolve this? We can't have any node just decide to do an update because we'd quickly run into inconsistency hell as we try to figure out how to get Boston to store antimatter widgets. One of the most straightforward approaches is *Leader and Followers*, where one of the nodes is marked as the leader, and the others are considered followers. In this situation, the leader handles all updates and broadcasts those updates to the followers. Let's say Neptune is the leader in this cluster. Then, Jupiter will forward Alice's A1 request to Neptune (Figure 2.7).

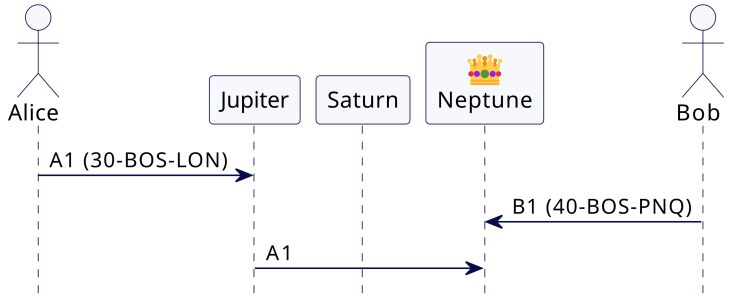

Figure 2.7 *Leader handling all the updates*

Neptune now gets both update requests, so it has the sole discretion as to how to deal with them. It can process the first one it receives (Bob's B1) and reject A1 (Figure 2.8).

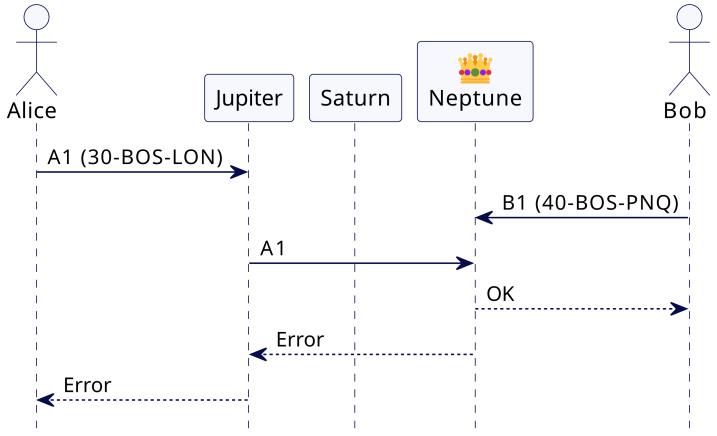

Figure 2.8 *Leader rejecting requests for insufficient widgets*

Dealing with the Leader Failing

That's what happens most of the time—when all goes well. But the point of getting a distributed system to work is what happens when things don't go well. Here's a different case. Neptune receives B1 and sends out its replication messages. But it is unable to contact Saturn. It could replicate only to Jupiter. At this point it loses all connectivity with the other two nodes. This leaves Jupiter and Saturn connected together, but disconnected from their leader (Figure 2.9).

So now what do these nodes do? For a start, how do they even find out what's broken? Neptune can't send Jupiter and Saturn a message saying the connection is broken . . . because the connection is broken. Nodes need a way to find out when connections to their colleagues break. They do this with a *HeartBeat*—or, more strictly, with the absence of a heartbeat.

A heartbeat is a regular message sent between nodes, just to indicate they are alive and communicating. Heartbeat does not necessarily require a distinct message type. When cluster nodes are already engaged in communication, such as when replicating data, the existing messages can serve the purpose of heartbeats. If Saturn doesn't receive a heartbeat from Neptune for a period of time, Saturn marks Neptune as down. Since Neptune is the leader, Saturn now calls for an election for a new leader (Figure 2.10).

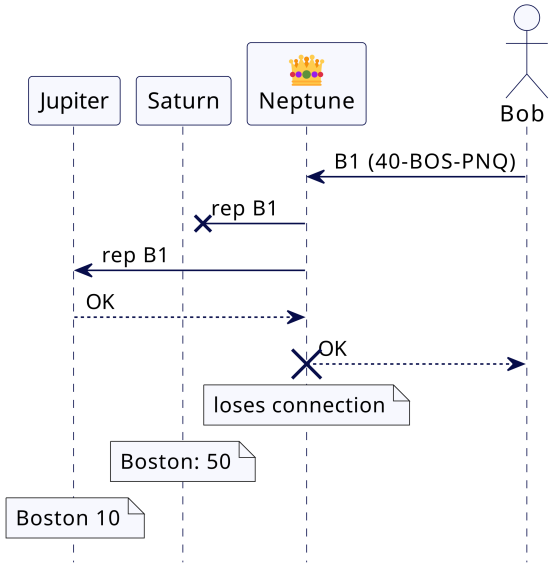

Figure 2.9 *Leader failure*

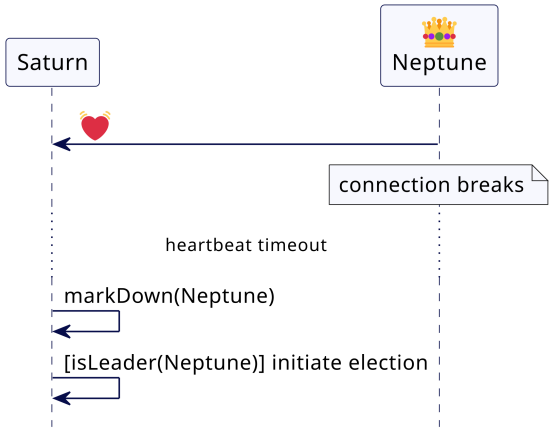

Figure 2.10 *Leader sending heartbeats*

The heartbeat gives us a way to know that Neptune has disconnected, so now we can turn to the problem of how to deal with Bob's request. We need to ensure that once Neptune has confirmed the update to Bob, even if Neptune crashes, the followers can elect a new leader with B1 applied to their data. But we also need to deal with more complication than that, as Neptune may have received multiple messages. Consider the case where there are messages from both Alice (A1) and Bob (B1) handled by Neptune. Neptune successfully replicates them both with Jupiter but is unable to contact Saturn before it crashes, as shown in Figure 2.11.

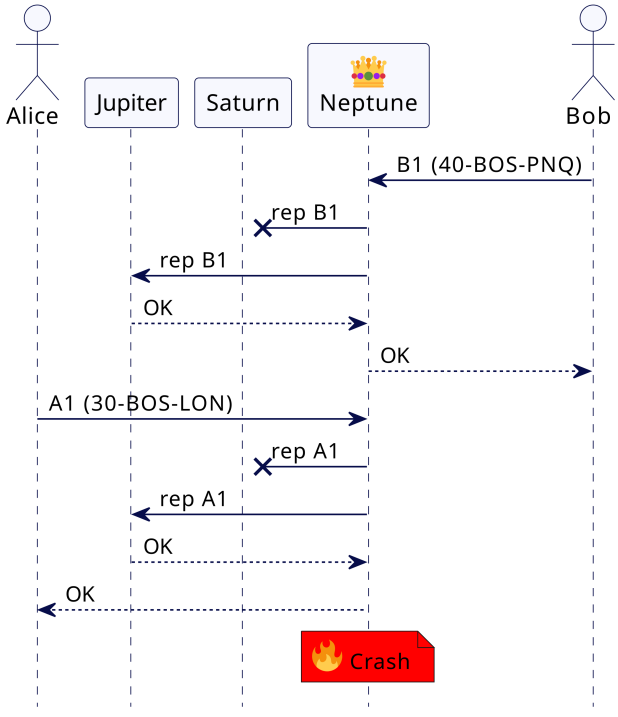

Figure 2.11 *Leader failure—incomplete replication*

In this case, how do Jupiter and Saturn deal with the fact that they have different states?

The answer is essentially the same as discussed earlier for resilience on a single node. If Neptune writes changes into a *Write-Ahead Log* and treats replication as

copying those log entries to its followers, then its followers will be able to figure out what the correct state is by examining the log entries (Figure 2.12).

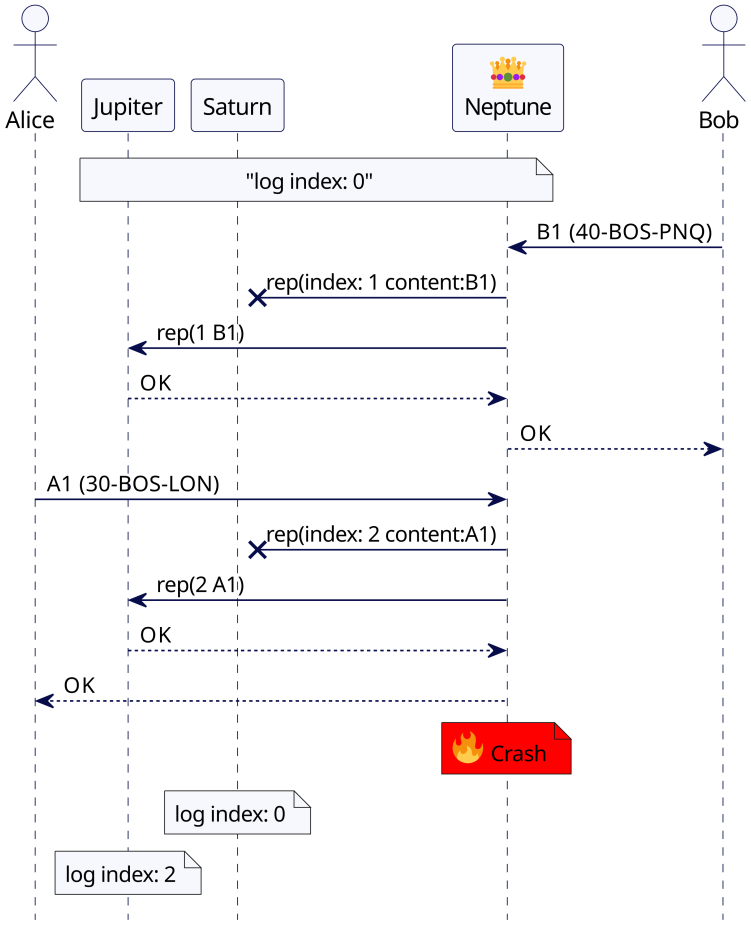

Figure 2.12 *Leader failure—incomplete replication—using log*

When Jupiter and Saturn elect a new leader, they can tell that Jupiter's log has later index entries, and Saturn can apply those log entries to itself to gain a consistent state with Jupiter.

This is also why Neptune can reply to Bob that the update was accepted, even though it hadn't heard back from Saturn. As long as a *Majority Quorum*—that is, a majority—of the nodes in the cluster have successfully replicated the log messages, Neptune can be sure that the cluster will maintain consistency even if the leader disconnects.

Multiple Failures Need a Generation Clock

We assumed here that Jupiter and Saturn can figure out whose log is most up to date. But things can get trickier. Let's say Neptune accepted a request from Bob to move 40 widgets from Boston to Pune but failed before replicating it (Figure 2.13).

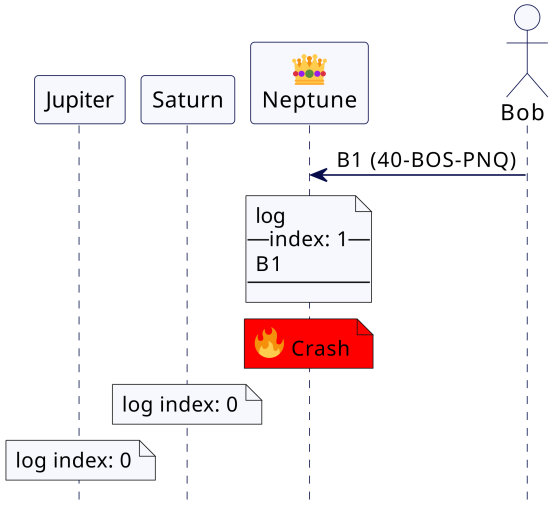

Figure 2.13 *Leader fails before replication.*

Jupiter is elected as a new leader, and accepts a request from Alice to move 30 widgets from Boston to London. But it also crashes before replicating the request to other nodes (Figure 2.14).

In a while, Neptune and Jupiter come back, but before they can talk, Saturn crashes. Neptune is elected as a leader. Neptune checks with itself and Jupiter for the log entries. It will see two separate requests at index 1, the one from Bob which it had accepted and the one from Alice that Jupiter has accepted. Neptune can't tell which one it should pick (Figure 2.15).

To solve this kind of situation, we use a *Generation Clock*. This is a number that increments with each leadership election. It is a key requirement of *Leader and Followers*.

Looking at the previous scenario again, Neptune was leader for generation 1. It adds Bob's entry in its log marking it with its generation (Figure 2.16).

When Jupiter gets elected as a leader, it increments the generation to 2. So when it adds Alice's entry to its log, it's marked for generation 2 (Figure 2.17).

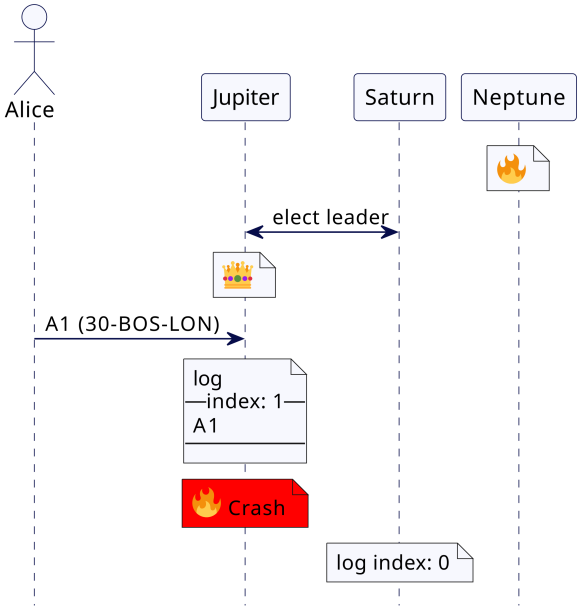

Figure 2.14 *New leader fails before replication.*

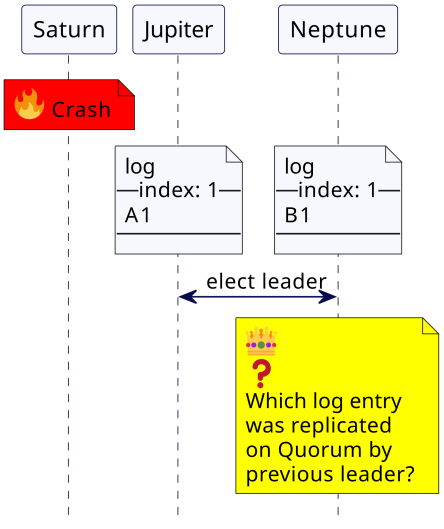

Figure 2.15 *Leader needs to resolve existing log entries.*

```
    public List<WatchEvent> getEvents(String key, Long stateChangesSince) {
        return this.events.stream()
                .filter(e -> e.getIndex() > stateChangesSince
                            && e.getKey().equals(key))
                .collect(Collectors.toList());
    }
}
```

When the client reestablishes the connection and resets watches, the events can be sent from history.

```
private void sendEventsFromHistory(String key, long stateChangesSince) {
    var events = eventHistory.getEvents(key, stateChangesSince);
    for (WatchEvent event : events) {
        notify(event, event.getKey());
    }
}
```

Using Multiversion Storage

To keep track of all the changes, it is possible to use multiversion storage. It keeps track of all the versions for every key and can easily get all the changes from the version asked for.

etcd from version 3 onward uses this approach.

Examples

- Apache ZooKeeper has the ability to set up watches on nodes. This is used by products like Apache Kafka for group membership and failure detection of cluster members.
- etcd has a watch implementation that is heavily used by Kubernetes for its resource watch implementation.[95]

95. https://kubernetes.io/docs/reference/using-api/api-concepts