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

FREE SAMPLE CHAPTER

Patterns of Distributed Systems

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.

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact intlcs@pearson.com.

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 .... *xvii*

Preface .... *xix*

Acknowledgments .... *xxiii*

About the Author .... *xxv*

Part I: Narratives *1*

Chapter 1: The Promise and Perils of Distributed Systems 3

The Limits of a Single Server 3

Separate Business Logic and Data Layer 5

Partitioning Data 6

A Look at Failures 7

Replication: Masking Failures 9

Process Crash 9

Network Delay 9

Process Pause 9

Unsynchronized Clocks 10

Defining the Term "Distributed Systems" 10

The Patterns Approach 10

Chapter 2: Overview of the Patterns 13

Keeping Data Resilient on a Single Server 14

Competing Updates 15

Dealing with the Leader Failing 17

Multiple Failures Need a Generation Clock 21

Log Entries Cannot Be Committed until They Are Accepted by a Majority Quorum 26

Followers Commit Based on a High-Water Mark 29

Leaders Use a Series of Queues to Remain Responsive to Many Clients ... 34
Followers Can Handle Read Requests to Reduce Load on the Leader ... 40
A Large Amount of Data Can Be Partitioned over Multiple Nodes ... 42
Partitions Can Be Replicated for Resilience ... 45
A Minimum of Two Phases Are Needed to Maintain Consistency across Partitions ... 46
In Distributed Systems, Ordering Cannot Depend on System Timestamps ... 49
A Consistent Core Can Manage the Membership of a Data Cluster ... 58
Gossip Dissemination for Decentralized Cluster Management ... 62

Part II: Patterns of Data Replication ... 69

Chapter 3: Write-Ahead Log ... 71
Problem ... 71
Solution ... 71
Implementation Considerations ... 73
Usage in Transactional Storage ... 74
Compared to Event Sourcing ... 76
Examples ... 76

Chapter 4: Segmented Log ... 77
Problem ... 77
Solution ... 77
Examples ... 79

Chapter 5: Low-Water Mark ... 81
Problem ... 81
Solution ... 81
Snapshot-Based Low-Water Mark ... 82
Time-Based Low-Water Mark ... 83
Examples ... 83

Chapter 6: Leader and Followers ... 85
Problem ... 85
Solution ... 85
Leader Election ... 86
Why Quorum Read/Writes Are Not Enough for Strong Consistency Guarantees ... 91
Examples ... 92

Chapter 7: HeartBeat 93
Problem 93
Solution 93
Small Clusters: Consensus-Based Systems 95
Technical Considerations 96
Large Clusters: Gossip-Based Protocols 97
Examples 98

Chapter 8: Majority Quorum 99
Problem 99
Solution 100
Deciding on Number of Servers in a Cluster 100
Flexible Quorums 101
Examples 102

Chapter 9: Generation Clock 103
Problem 103
Solution 104
Examples 107

Chapter 10: High-Water Mark 109
Problem 109
Solution 109
Log Truncation 112
Examples 115

Chapter 11: Paxos 117
Problem 117
Solution 117
Flow of the Protocol 118
An Example Key-Value Store 127
Flexible Paxos 132
Examples 132

Chapter 12: Replicated Log 133
Problem 133
Solution 133
Multi-Paxos and Raft 134
Replicating Client Requests 135
Leader Election 141

Technical Considerations 150
Push vs. Pull 151
What Goes in the Log? 151
Examples 158

Chapter 13: Singular Update Queue 159
Problem 159
Solution 159
Choice of the Queue 164
Using Channels and Lightweight Threads 164
Backpressure 165
Other Considerations 166
Examples 166

Chapter 14: Request Waiting List 167
Problem 167
Solution 167
Expiring Long Pending Requests 172
Examples 173

Chapter 15: Idempotent Receiver 175
Problem 175
Solution 175
Expiring the Saved Client Requests 179
Removing the Registered Clients 180
At-Most-Once, At-Least-Once, and Exactly-Once Actions 181
Examples 181

Chapter 16: Follower Reads 183
Problem 183
Solution 183
Finding the Nearest Replica 184
Disconnected or Slow Followers 187
Read Your Own Writes 188
Linearizable Reads 191
Examples 191

Chapter 17: Versioned Value 193
Problem 193

Solution 193
Ordering of Versioned Keys 194
Reading Multiple Versions 197
MVCC and Transaction Isolation 199
Using RocksDB-Like Storage Engines 200
Examples 201

Chapter 18: Version Vector 203
Problem 203
Solution 203
Comparing Version Vectors 205
Using Version Vector in a Key-Value Store 207
Examples 216

Part III: Patterns of Data Partitioning 217

Chapter 19: Fixed Partitions 219
Problem 219
Solution 220
Choosing the Hash Function 221
Mapping Partitions to Cluster Nodes 222
Alternative Solution: Partitions Proportional to Number of Nodes 236
Examples 241

Chapter 20: Key-Range Partitions 243
Problem 243
Solution 244
Predefining Key Ranges 244
An Example Scenario 247
Auto-Splitting Ranges 249
Examples 255

Chapter 21: Two-Phase Commit 257
Problem 257
Solution 257
Locks and Transaction Isolation 261
Commit and Rollback 268
An Example Scenario 273

Using Versioned Value 279
Using Replicated Log 291
Failure Handling 291
Transactions across Heterogeneous Systems 297
Examples 297

Part IV: Patterns of Distributed Time 299

Chapter 22: Lamport Clock 301
Problem 301
Solution 301
Causality, Time, and Happens-Before 302
An Example Key-Value Store 303
Partial Order 305
A Single Leader Server Updating Values 306
Examples 307

Chapter 23: Hybrid Clock 309
Problem 309
Solution 309
Multiversion Storage with Hybrid Clock 312
Using Timestamp to Read Values 314
Assigning Timestamp to Distributed Transactions 314
Examples 316

Chapter 24: Clock-Bound Wait 317
Problem 317
Solution 318
Read Restart 322
Using Clock-Bound APIs 325
Examples 332

Part V: Patterns of Cluster Management 335

Chapter 25: Consistent Core 337
Problem 337
Solution 337
Metadata Storage 339
Handling Client Interactions 339
Examples 342

Chapter 26: Lease 345
Problem 345
Solution 345
Attaching the Lease to Keys in the Key-Value Storage 351
Handling Leader Failure 353
Examples 354

Chapter 27: State Watch 355
Problem 355
Solution 355
Client-Side Implementation 356
Server-Side Implementation 356
Handling Connection Failures 359
Examples 362

Chapter 28: Gossip Dissemination 363
Problem 363
Solution 363
Avoiding Unnecessary State Exchange 368
Criteria for Node Selection to Gossip 371
Group Membership and Failure Detection 372
Handling Node Restarts 372
Examples 373

Chapter 29: Emergent Leader 375
Problem 375
Solution 375
Sending Membership Updates to All the Existing Members 379
An Example Scenario 382
Handling Missing Membership Updates 384
Failure Detection 385
Comparison with Leader and Followers 392
Examples 392

Part VI: Patterns of Communication between Nodes 393

Chapter 30: Single-Socket Channel 395
Problem 395
Solution 395
Examples 397

Chapter 31: Request Batch ... **399**
Problem ... 399
Solution ... 399
Technical Considerations ... 404
Examples ... 404

Chapter 32: Request Pipeline ... **405**
Problem ... 405
Solution ... 405
Examples ... 408

References ... ***409***

Index ... ***413***

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.

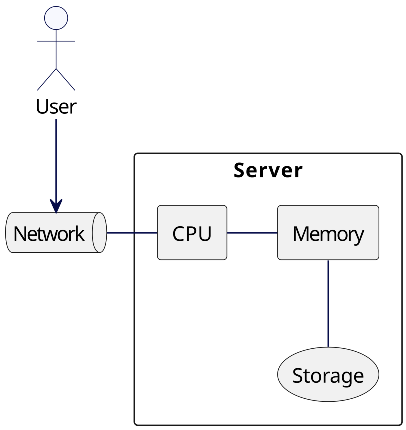

Figure 1.1 *Resources of computation*

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.

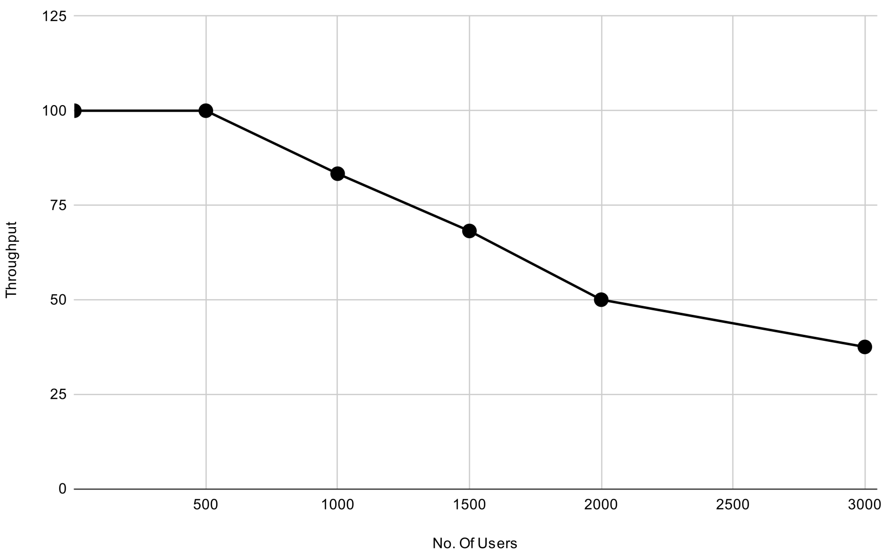

Figure 1.2 *Drop in throughput with increase in requests*

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).

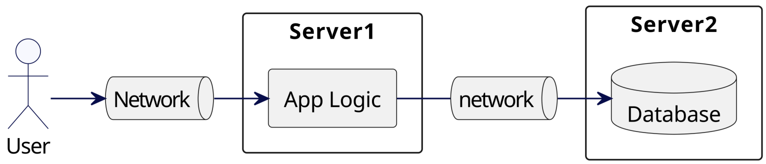

Figure 1.3 *Separating compute and data*

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).

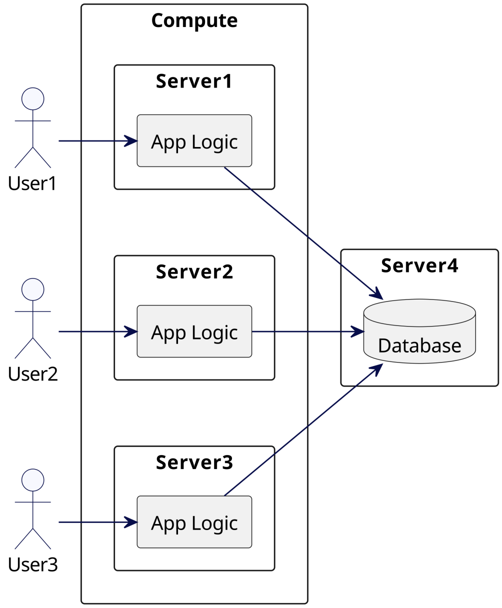

Figure 1.4 *Scaling compute with multiple servers*

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.

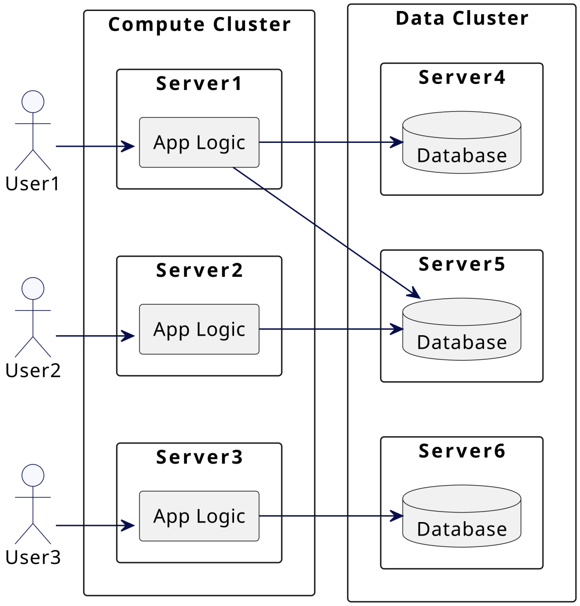

Figure 1.5 *Scaling data by distributing on multiple servers*

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`

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).

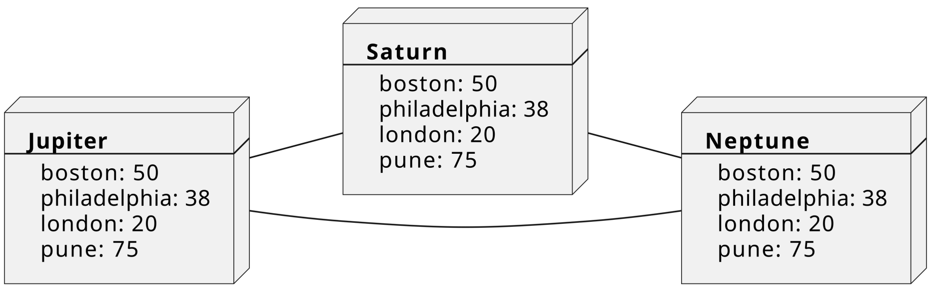

Figure 2.2 *Replicated data record*

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).

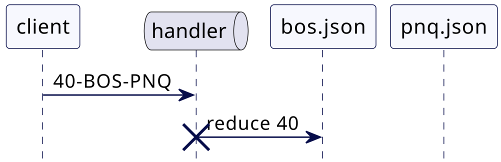

Figure 2.3 *Node crash causes inconsistency*

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

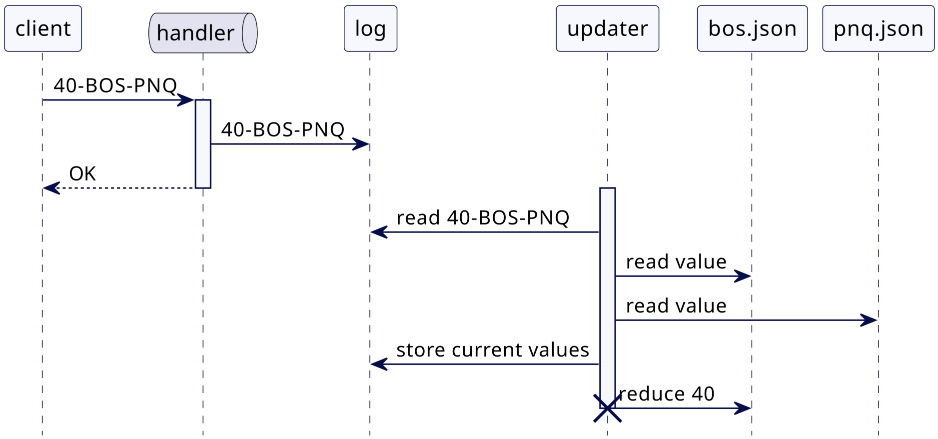

Figure 2.4 *Using WAL*

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.)

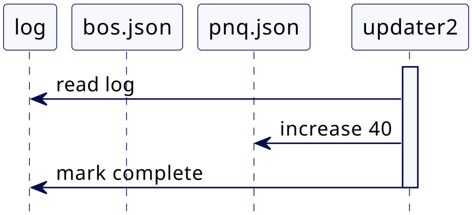

Figure 2.5 *Recovery using WAL*

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).

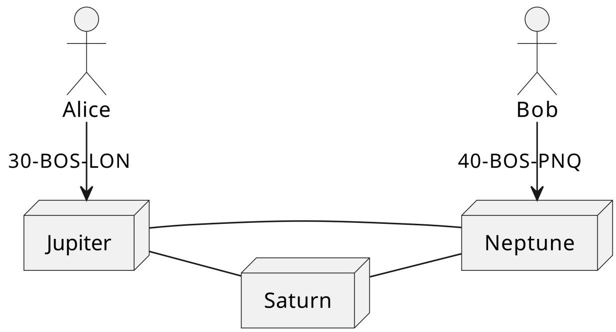

Figure 2.6 *Competing updates*

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).

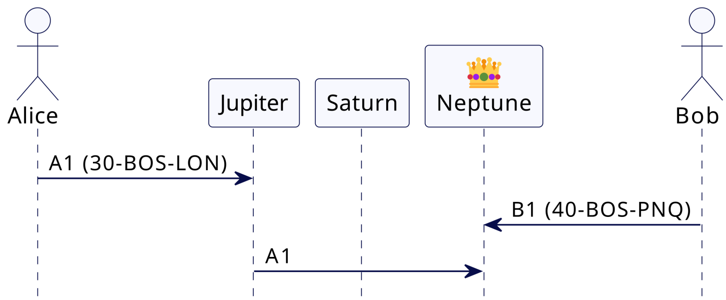

Figure 2.7 *Leader handling all the updates*

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).

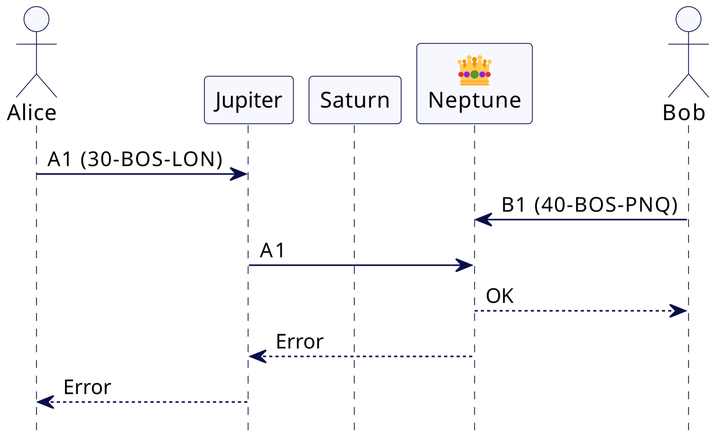

Figure 2.8 *Leader rejecting requests for insufficient widgets*

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).

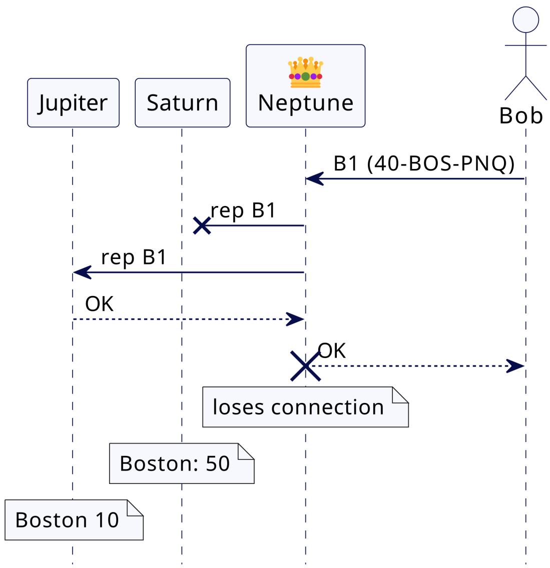

Figure 2.9 *Leader failure*

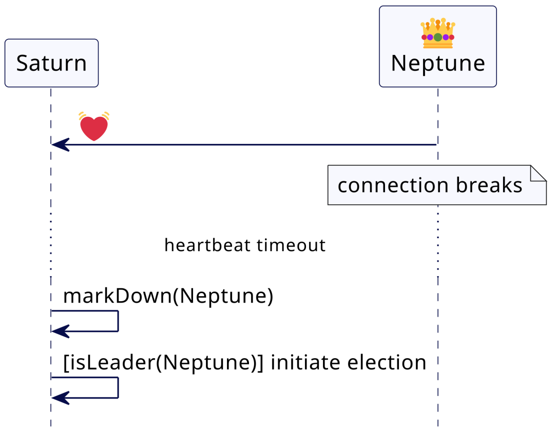

Figure 2.10 *Leader sending heartbeats*

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.

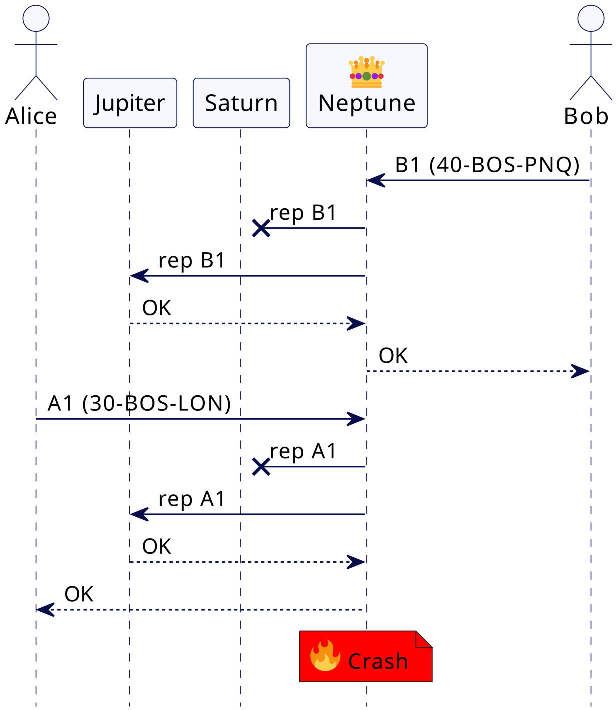

Figure 2.11 *Leader failure—incomplete replication*

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).

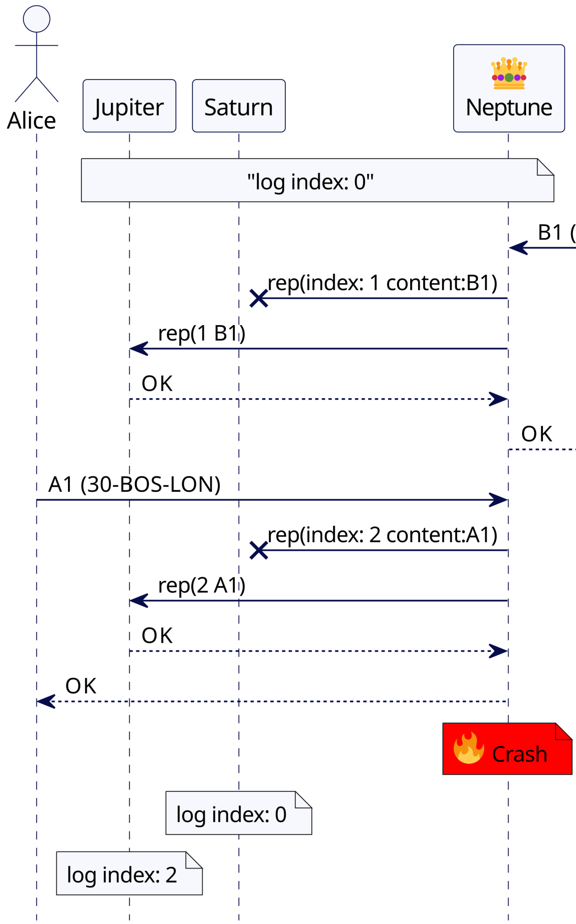

Figure 2.12 *Leader failure—incomplete replication—using log*

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).

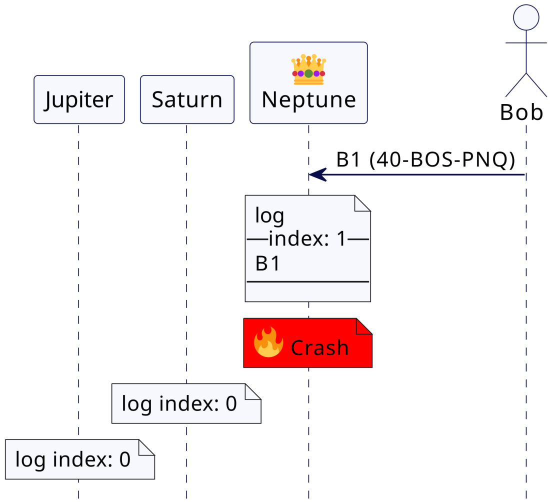

Figure 2.13 *Leader fails before replication.*

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).

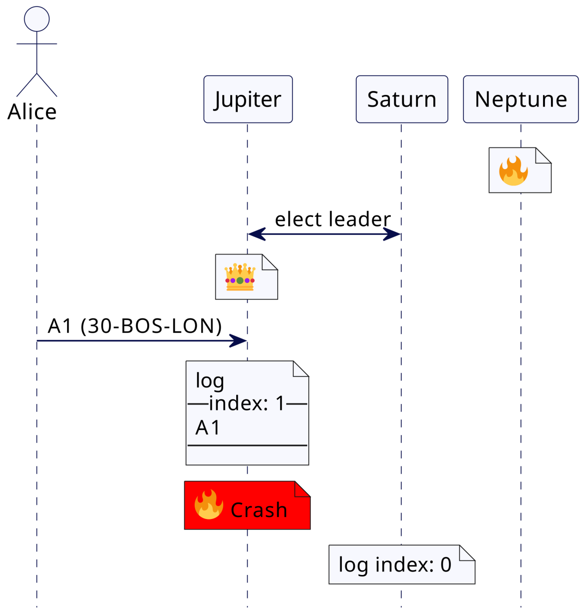

Figure 2.14 *New leader fails before replication.*

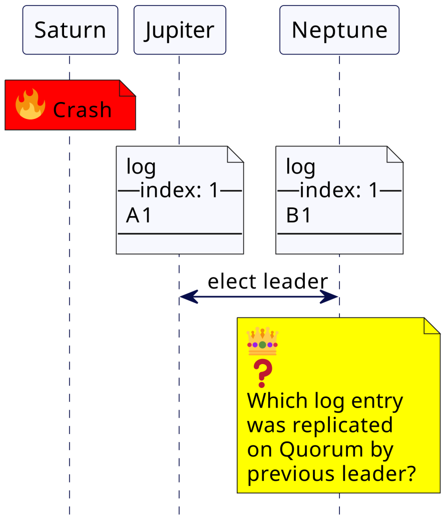

Figure 2.15 *Leader needs to resolve existing log entries.*

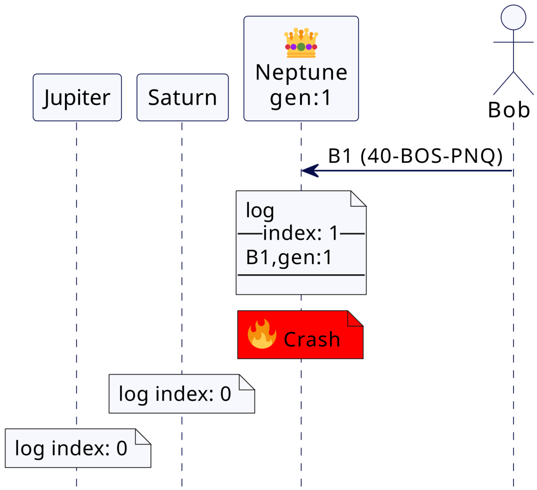

Figure 2.16 *Leader adds generation to log entries.*

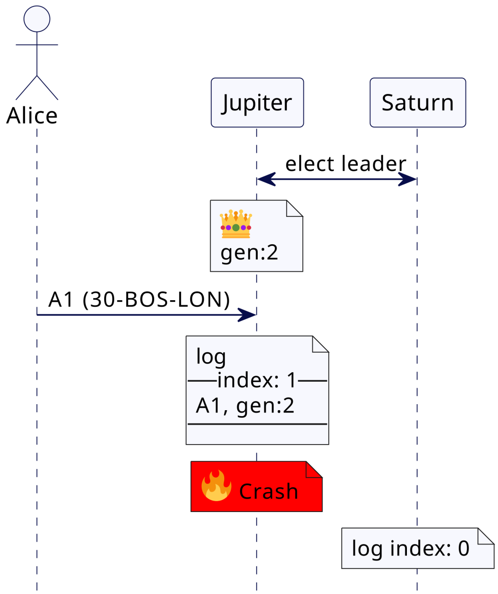

Figure 2.17 *New leader increments generation.*

Now, when Neptune is again elected as a leader, it will be for generation 3. Before it starts serving the client requests, it checks the logs of all the available nodes for entries which are not replicated on the *Majority Quorum*. We call these entries as "uncommitted," as they are not yet applied to data. We will see how each node figures out which entries are incompletely replicated in a while. But once the leader knows about these entries, it completes the replication for those entries. In case of conflict, it safely picks up the entry with higher generation (Figure 2.18).

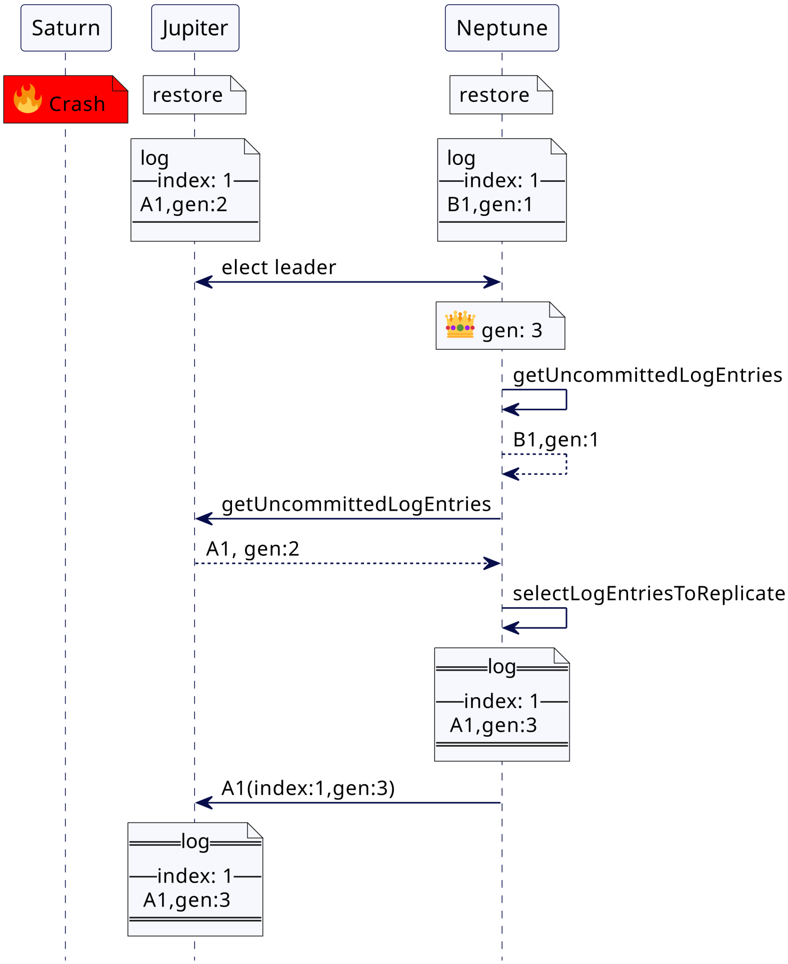

Figure 2.18 *Conflicting log entries are resolved based on generation.*

After selecting the entry with the latest generation, Neptune overwrites the uncommitted entry in its own log with its current generation number and replicates with Jupiter.

Every node tracks the latest generation it knows of the leader. This is helpful in another problem that might occur, as Figure 2.19 demonstrates. When Jupiter became leader, the previous leader, Neptune, might not have crashed, but just temporarily disconnected. It might come back online and send the requests to Jupiter and Saturn. If Jupiter and Saturn have elected a new leader and accepted

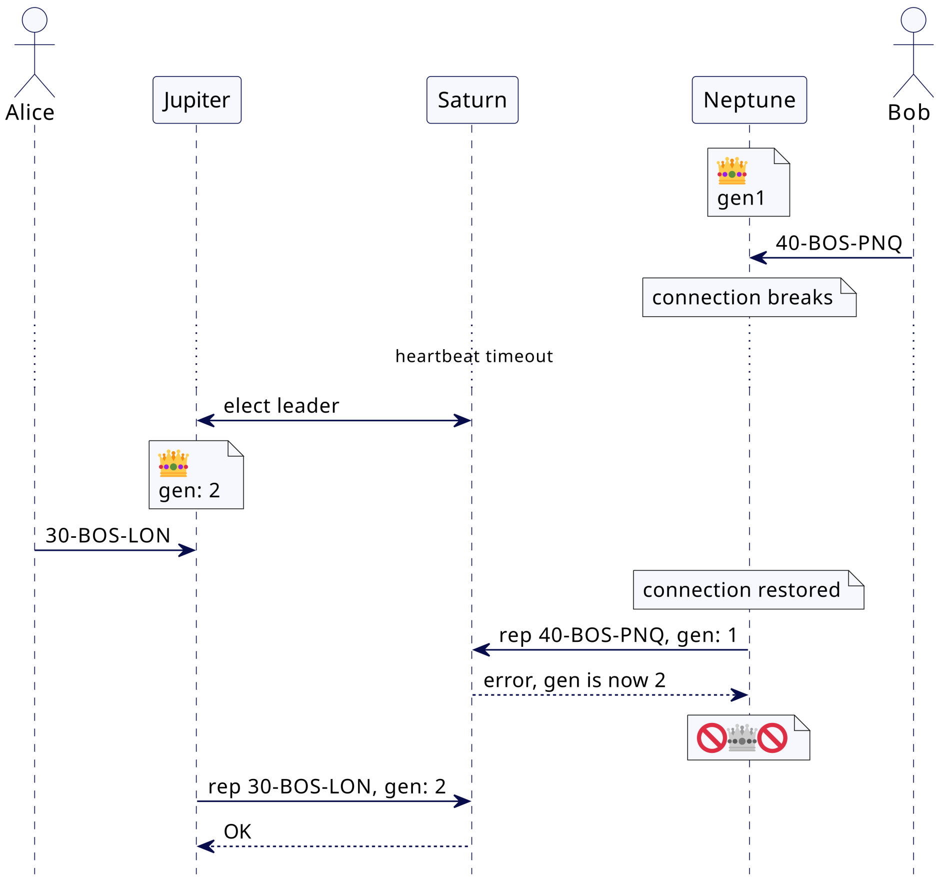

Figure 2.19 *Generation helps detecting stale requests from old leader.*

requests from Alice, what should they do when they suddenly start getting requests from Neptune? Generation Clock is useful in this case as well. Every request is sent to cluster nodes, along with the generation clock. So every node can always choose the requests with the higher generation and reject the ones with the lower generation.

Log Entries Cannot Be Committed until They Are Accepted by a Majority Quorum

As seen above, entries like B1 can be overwritten if they haven't been successfully replicated to a *Majority Quorum* of nodes in the cluster. So the leader cannot apply the request to its data store after just appending to its own log—it has to wait until it gets enough acknowledgments from other nodes first. When an update is added to a local log, it is **uncommitted**, until the leader has had replies from a Majority Quorum of other nodes, at which point it becomes **committed**. In the case of the example above, Neptune cannot commit B1 until it hears that at least one other node has accepted it, at which point that other node, plus Neptune itself, makes two out of three nodes—a majority and thus a Majority Quorum.

When Neptune, the leader, receives an update, either from a user (Bob) directly or via a follower, it adds the uncommitted update to its log and then sends replication messages to the other nodes. Once Saturn (for example) replies, that means two nodes have accepted the update, Neptune and Saturn. This is two out of three nodes, which is the majority and thus a Majority Quorum. At that point Neptune can commit the update (Figure 2.20).

The importance of the Majority Quorum is that it applies to decision by the cluster. Should a node fail, any leadership election must involve a Majority Quorum of nodes. Since any committed updates have also been sent to a Majority Quorum of nodes, we can be sure that committed updates will be visible during the election.

If Neptune receives Bob's update (B1), replicates, gets an acknowledgment from Saturn, and then crashes, Saturn still has a copy of B1. If the nodes then elect Jupiter as the leader, Jupiter must apply any uncommitted updates—that is, B1—before it can start accepting new ones (Figure 2.21).

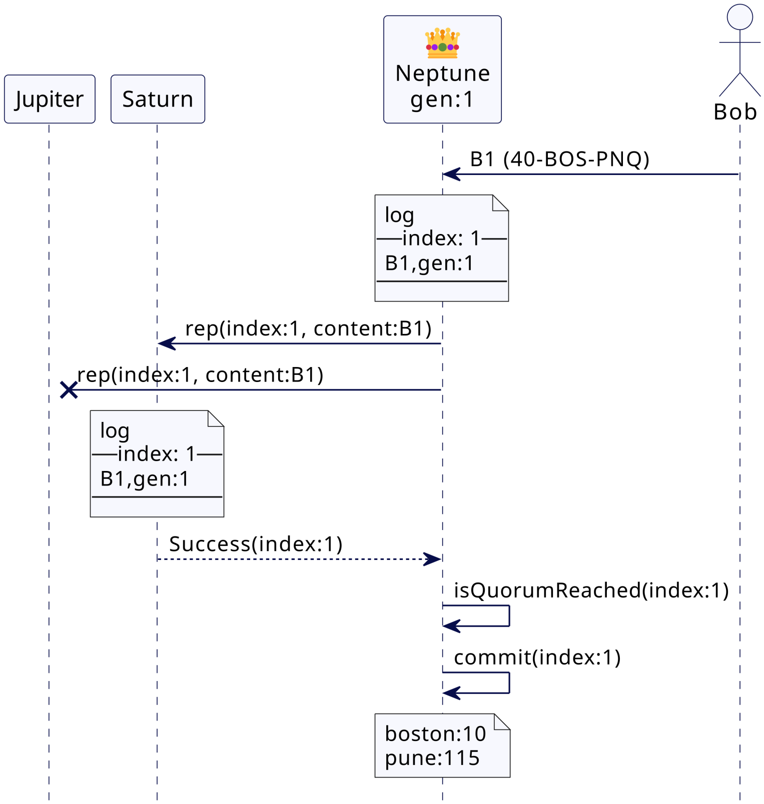

Figure 2.20 *Log entries are committed once they are accepted by a Majority Quorum.*

When the log is large, moving the log across nodes for leader election can be costly. The most commonly used algorithm for *Replicated Log*, Raft [Ongaro2014], optimizes this by electing the leader with the most up-to-date log. In the above example this would elect Saturn as the leader.

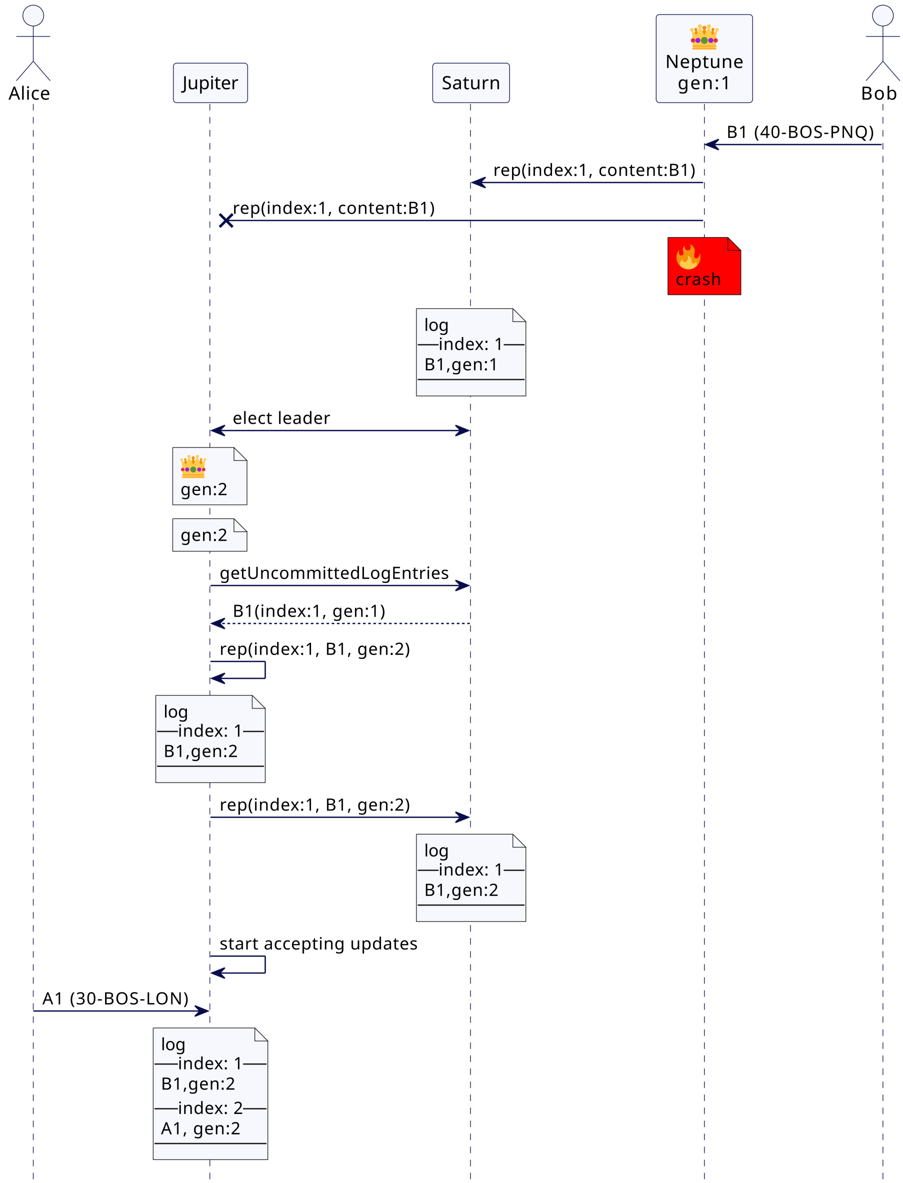

Figure 2.21 *New leader commits uncommitted log entries.*

Followers Commit Based on a High-Water Mark

As we've seen, leaders commit when they get acknowledgments from a *Majority*

Quorum—but when do followers commit their log entries? In the three node example we've been using, it's obvious. Since we know the leader must have added the log entry before it replicates, any node knows that it can commit since it and the leader form a Majority Quorum. But that isn't true for larger clusters. In a five-node cluster, a single follower and a leader are only two of five.

A *High-Water Mark* solves this conundrum. Simply put, the High-Water Mark is maintained by the leader and is equal to the index of the latest update to be committed. The leader then adds the High-Water Mark to its *HeartBeat*. Whenever a follower receives a HeartBeat, it knows it can commit all its log entries up to the High-Water Mark.

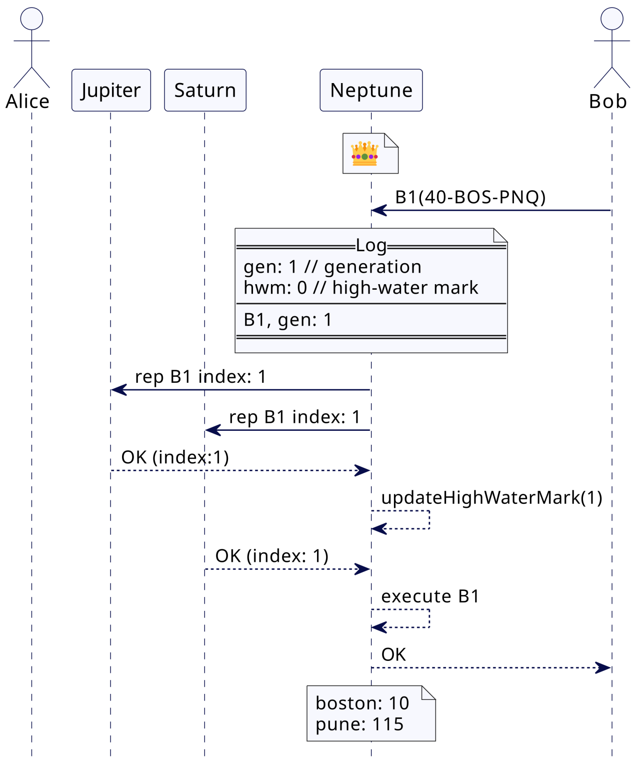

Figure 2.22 *Leader tracks High-Water Mark.*

Let's look at an example of this (Figure 2.22). Bob sends a request (B1) to Neptune. Neptune replicates the request to Jupiter and Saturn. Jupiter acknowledges first, allowing Neptune to increase its High-Water Mark to 1, execute the update against its data store, and return success to Bob. Saturn's acknowledgment is late, and since it's not higher than the High-Water Mark, Neptune takes no action on it.

Neptune now gets three requests from Alice (A1, A2, and A3). Neptune puts all of these into its log and starts sending replication messages. The link between Neptune and Saturn, however, gets tangled and Saturn doesn't get them (Figure 2.23).

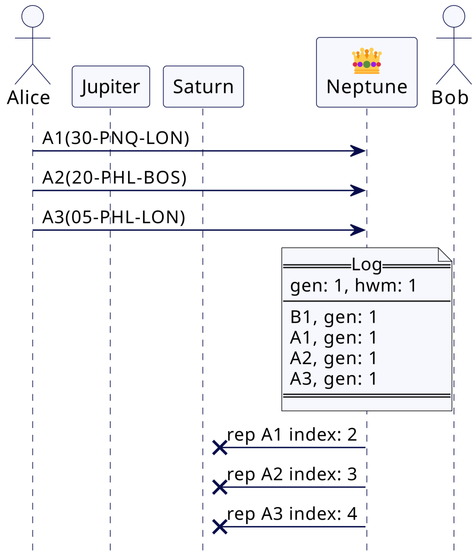

Figure 2.23 *Nodes missing replication of log entries*

After the first two messages, Neptune coincidentally sends out heartbeats, which alert followers to update their High-Water Mark. Jupiter acknowledges A1, allowing Neptune to update its High-Water Mark to 2, execute the update, and notify Alice. But then Neptune crashes before it's able to replicate A3, as shown in Figure 2.24.

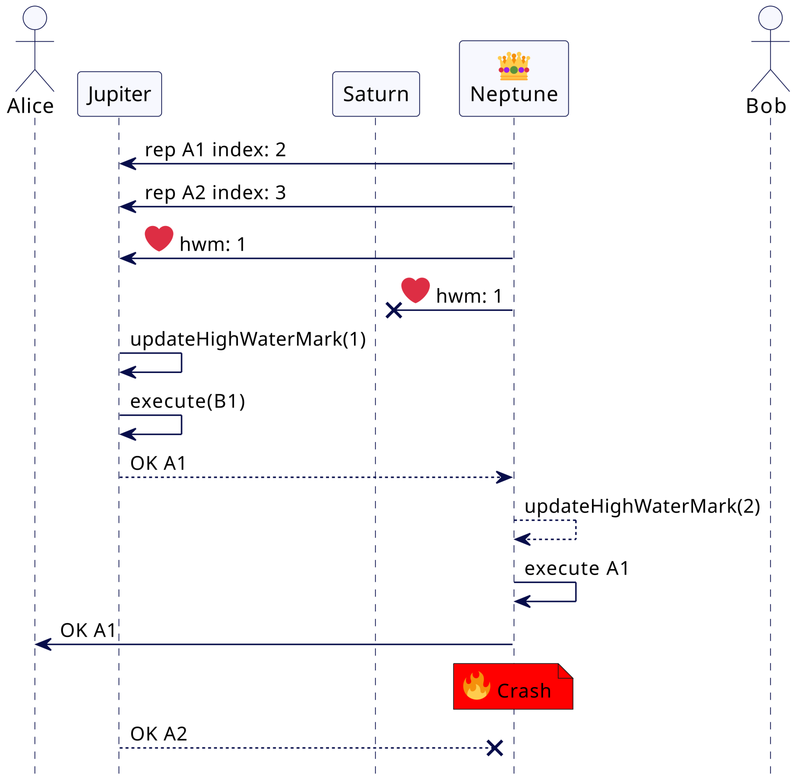

Figure 2.24 *The High-Water Mark is propagated using HeartBeat.*

At this point, here are the states of the nodes:

	Jupiter	Saturn	Neptune
gen	1	1	1
hwm	1	0	2
log	B1 A1 A2	B1	B1 A1 A2 A3

Jupiter and Saturn fail to get HeartBeat from Neptune and thus hold an election for a new leader. Jupiter wins and gathers log entries. In doing this it accepts that A2 reached Majority Quorum and sets its High-Water Mark to 3. Jupiter replicates its log to Saturn, and when Saturn gets a HeartBeat with High-Water

Mark of 3 it's able to update its High-Water Mark and execute the updates against its store (Figure 2.25).

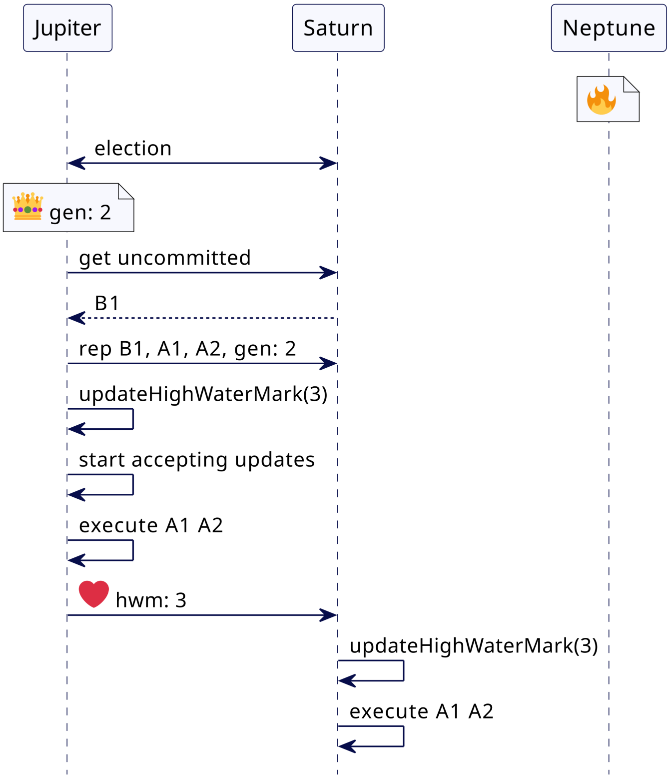

Figure 2.25 *New leader replicates missing log entries and High-Water Mark.*

Now, the state of the nodes is:

	Jupiter	**Saturn**	**Neptune**
gen	2	2	1
hwm	3	3	2
log	B1 A1 A2	B1 A1 A2	B1 A1 A2 A3

At this point Alice times out of her A3 request and resends it (A3.2), which routes to Jupiter as the new leader. Just as this happens, Neptune starts back up again. Neptune tries to replicate A3, and is told that there's a new generation of leader, so Neptune accepts that it's now a follower of Jupiter and discards its log down to its High-Water Mark. Jupiter sends replication messages for A2 and A3.2. Once Jupiter gets an acknowledgment for A3.2, it can update its High-Water Mark, execute the update, and respond to Alice (Figure 2.26).

Saturn and Neptune will update their states on the next HeartBeat from Jupiter.

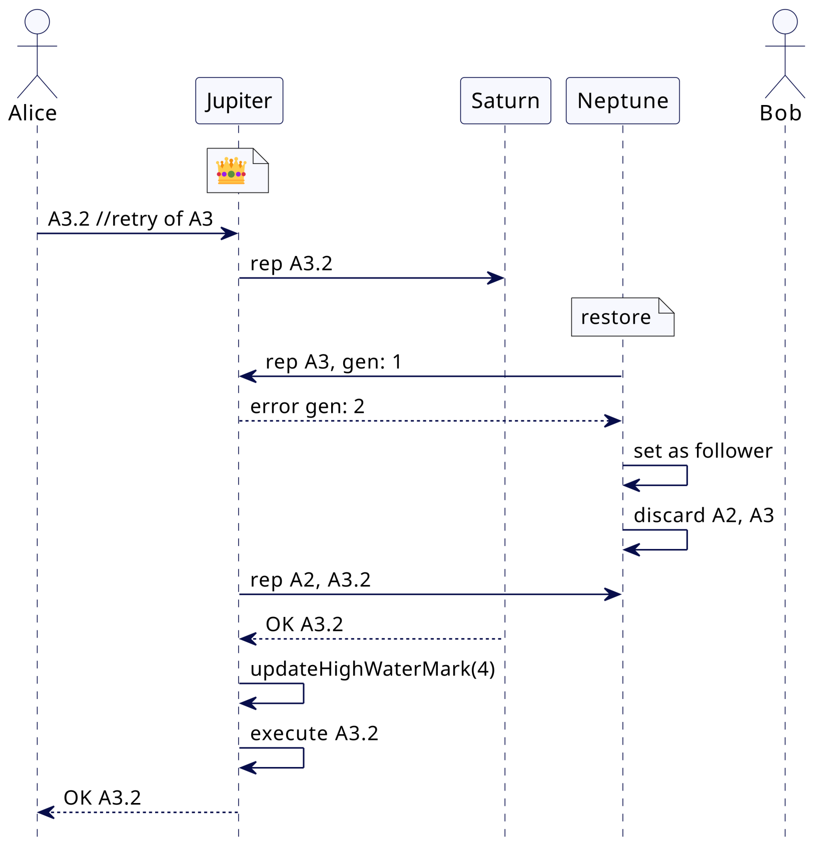

Figure 2.26 *Old leader discards conflicting log entries.*

Index

A

acceptors (Paxos), 118–119
acknowledgment messages, 379
ActiveM message broker, 297
Akka toolkit, 11
  coordinators in, 222
  data clusters in, 68
  discovery protocols in, 376, 392
  failure detection in, 97–98, 386
  gossip protocols in, 381
  partitioning in, 43, 221, 241
  sending heartbeats asynchronously in, 96
  split brain resolver in, 388
all-to-all heartbeating, 386
Amazon
  AWS (Amazon Web Services), 3
    Clock Bound library, 58, 322, 325, 333
    Time Sync Service, 322, 325, 333
  S3 (Simple Storage Service), 188
Apache
  BookKeeper, 116, 343, 404
  Cassandra, 11, 58, 62, 66, 68, 79, 91, 96–98, 102, 107, 128, 132, 166, 173, 212, 216, 239, 371, 373
  Flink, 343
  HBase, 44, 181, 250, 255
  HDFS, 343
  Ignite, 43, 221, 225, 241, 386, 392
  Kafka, 11, 39, 46, 58–59, 61, 76, 79, 83, 92, 107, 116, 151, 158, 166, 173, 181, 187, 192, 220, 222, 225, 229, 241, 297, 340, 343, 349, 354, 360, 362, 398, 404, 408
  Pulsar, 11
  Spark, 343
  ZooKeeper, 11, 39, 59, 61, 82–83, 87, 90–92, 95, 100, 107, 152–153, 166, 181, 229, 339–340, 343, 349, 351, 354, 361–362, 397
`ArrayBlockingQueue` class (Java), 164–165
asynchronous communication, 167
at-least-once, at-most-once actions, 181
atomic operations, 74
auto-splitting, 249–255
AWS (Amazon Web Services), 3
  Clock Bound library, 58, 322, 325, 333
  Time Sync Service, 322, 325, 333
Azure platform (Microsoft), 3
  Cosmos DB, 128, 134

B

backpressure, 165, 358
bandwidth, 3, 97, 393
Birman, Kenneth, 100
blockchains, 151, 158, 373
blocking queues, 407
Bolt database, 194, 200
BookKeeper service (Apache), 116
  metadata in, 343
  request batching in, 404
bookmarks, in write operations, 191
bottlenecks
  in logs, 74, 77
  in partitioning, 229
  on leaders, 341
brokers
  caching metadata across, 229
  controller, 59, 92, 166
  follower, 92, 192
  leader, 92
  message, 134
Byzantine faults, 134

C

callbacks, 35, 167–168
candidates, 86
CASPaxos register, 131

Cassandra database (Apache), 11
  data clusters in, 58, 62, 68
  durability of, 76
  failure detection in, 91, 97–98
  generations in, 107
  implementing Paxos, 128, 132
  log segmentation in, 79
  LWW conflict resolution in, 212, 216
  Majority Quorum in, 173
  metadata in, 66, 371, 373
  multiple random tokens in, 239
  node communication in, 173
  paused processes in, 96
  SEDA architecture in, 166
  updates in, 102
causal clusters, 191
causal consistency, 188
causal relationship, 302
Chubby lock service (Google), 339, 342, 354
clients
  communicating with leaders, 134, 340
  identifiers of, 175, 178, 215
  interactions of, 339
  log entries visible to, 112
  number of, 216
  read-only, 183
  receiving data from event history, 361
  refreshing metadata, 229
  registered with servers, 36, 175–177, 355–360
    removing, 180
  requests of:
    duplicated, 151
    executing, 388
    replicating, 135–140
    responding to, 175
  tracking queued requests, 400
Clock Bound library (AWS), 58, 322, 325, 333
clock skew, 56–58, 157, 290, 299
Clock-Bound Wait pattern, 299, 317–333
  APIs for, 325–331
  consistency in, 54–58, 317
  read restarts in, 322–324
  timestamps in, 290
`ClockErrorBound` (AWS), 325
clocks
  atomic, 325
  error bounds of, 325–326
  hybrid logical, 309–316, 318
  lagging behind, 319
  Lamport, 301–307
  monotonic, 154, 157, 346, 349
  not monotonic, 264, 286, 346
  time-of-the-day, 299
  unsynchronized, 10, 49, 299, 349
closed timestamps, 192
cloud services, 3
Cloudflare, 141
cluster controllers, 59
cluster time, 346
clusters
  causal, 191
  consensus-based, 95, 100, 141
  decision-making in, 26
  gossip-based, 97
  managing, 58–61
  registering, 222–225
  size of, 90, 100–102, 220, 337, 364, 386
  states in, 133, 372
  updates in, 16, 34, 100, 363
CockroachDB database, 11
  data clusters in, 58
  gossip protocol in, 371, 373
  Hybrid Clock in, 58, 279, 290, 309, 314, 316
  latencies in, 186
  MVCC backend in, 201, 307
  partitioning in, 44, 255, 297
  read restarts in, 333
  reading from follower servers in, 192
  serializable isolation in, 261
  timestamps in, 291, 314
  transactional intents in, 293
column-family databases, 212
Command pattern, 72
Commit Log. *See* Write-Ahead Log pattern
`commitIndex` (Raft), 115, 137–140, 148
commit-wait operation, 326
`compareAndSwap` operation, 90
concurrency, 199
`ConcurrentLinkedQueue` class (Java), 164
conflict resolvers, 211–213
connections
  closing, 357
  failures of, 127, 360
  pipelined, 358
  timeouts on, 397
  watch events on, 356–362
consensus algorithms, 117, 132
  building, 133–134
  determining up-to-date servers in, 87
  fault tolerance in, 337–338
  High-Water Mark in, 115
  leader elections in, 86, 90–92, 141
  liveness issues in, 338
  processing requests in, 166
  replication lagging in, 183
  snapshot mechanisms in, 82

consistency, 46–49, 75
causal, 188, 191
eventual, 62, 68, 372
external, 317
Consistent Core pattern, 337–343
built with state machine replication, 158
client registration with, 177
connection failures in, 360
finding leaders in, 151
key changes in, 355
leader elections in, 90–91, 353, 392
leases in, 345–353
partitioning data in, 43, 222
read requests in, 191
storing metadata with, 237, 239
supporting hierarchical storage in, 339, 359
tracking clusters in, 58–62
vs. Emergent Leader, 375
vs. Gossip Dissemination, 68, 372
Consul service (HashiCorp)
Consistent Core in, 372
data clusters in, 68
failure detection in, 97–98, 372
gossip protocol in, 372–373
leader elections in, 91
leases in, 158
sending heartbeats asynchronously in, 96
stale data in, 153
timeouts in, 157
controller brokers, 59, 92
Controller Quorum (Kafka), 360
Coordinated Universal Time. *See* UTC
coordinators, 46–49, 222–235, 375
choosing timestamps, 279
clashes between, 386
communicating the outcome of transactions, 291
crashing, 292
creating key ranges, 245, 249
designating, 68, 293
fault tolerance of, 297
mapping partitions to ranges, 247
performing rollbacks, 271–272, 292
receiving commit requests, 268–270
retrying connections periodically, 389
tracking:
received updates, 379
transactions, 258–260, 265, 295
Cosmos DB database (Microsoft Azure), 128, 134
counters, 49
for log entries, 193
for nodes, 203
for processes, 104
for versions, 306
CPUs (central processing units), 3–6
partition splitting and, 255
crashes, 9
conflicts after, 112, 115
detecting, 17–19
restarting after, 15
restoring states after, 73, 292
See also failures
CRC records, 74
crystal oscillators, 299, 301, 346

D

data
availability of, 100
after crashing, 9
clusters of. *See* clusters
consistency of, 85
distributing, 5, 8, 10, 13, 42
failures in, 7–9
fraudulent, 134
inconsistencies in, 283
integrity of, 9
mapping to nodes, 219–222
moving between nodes, 42–44, 230–233
partitioning, 6, 42–44, 217–297
replicating, 9, 13, 45, 69–216
resilient, 14–15
spreading, 62–63, 364
stale, 40, 153, 183, 188, 191, 299, 340
uncommitted, 26
updating:
by followers, 40–41
from multiple threads, 34
stopped, 187
data stores, 42
distributed, 286, 291, 297
locks in, 293
nonblocking, 199
partitioning in, 253
Replicated Log in, 152
serializable isolation in, 261–262
transaction restarting in, 264
databases, 5
column-family, 212
durability of, 76
failure assumptions in, 134
implementing transactions in, 15
partitioned, 151
read restarts in, 58
relational, 100
datacenters, 183

Date-Time API (Java), 325, 346
deadlocks, 262–263, 275
Dean, Jeff, 7
Dgraph database, 286
DHCP (Dynamic Host Configuration Protocol), 354
discovery protocols, 376
disks
  durability of, 74
  failures of, 7
  performance of, 3
  saving data to, 404
  slow, 187
distributed systems
  definition of, 3, 10
  liveness vs. safety of, 99
  scaling, 217
dotted version vectors, 216
duplicate detection, 342

E

elapsed time, 346
elections. *See* leaders, electing
`electionTimeout` interval, 157
embedded storage engines, 200
Emergent Leader pattern, 375–392
  data clusters in, 68
  split brain problem in, 386–390
  updates in:
    missing, 384
    sending, 379–383
  vs. Leader and Followers, 392
enterprise systems, failure assumptions in, 134
EPaxos algorithm, 151
ephemeral nodes, 90, 351
epidemics, 62–63, 364
epoch. *See* Generation Clock pattern
error-on-conflict policy, 263–266, 279
etcd key-value store, 11
  Bolt database in, 200
  channels in, 166
  coordinators in, 222
  data clusters in, 59, 61
  forwarding client requests in, 340
  goroutines in, 166
  leader elections in, 90–91
  leases in, 158, 354
  metadata in, 343
  processing requests in, 166
  read requests in, 152
  snapshot mechanisms in, 82
  stale data in, 153
  waiting lists in, 173
  watch channels in, 358, 362
etcd3 interface, 201
events
  order of, 104
  sourcing, 76
  storing history of, 361
`EvictingQueue` (Java), 361
exactly-once actions, 181
`ExecutorService` interface (Java), 160
external consistency, 317
external service calls, 166

F

failover protocol, 354
failures
  access to resources and, 345
  assuming, 134
  connection, 360
  detecting, 86, 91, 93–98, 153, 157, 222–224, 372, 378, 385–390
  handling, 291–292
  log files and, 74
  managing, 8, 17–26, 59, 270–272
  masking, 9
  multiple, 21–26
  probability of, 7–8
  recovering after, 109
  single point of, 287
  tolerating, 100–101, 134, 177, 297, 337–338, 345, 364
  See also crashes
Fixed Partitions pattern, 43–44, 219–235, 241
  choosing placement of, 392
  distributing across nodes, 219–220, 375
  tracking clusters in, 61
Flink framework (Apache), 343
FLP impossibility result, 127
flushing, 73–74
follower brokers, 92
Follower Reads pattern, 183–192
  replicating updates to followers in, 40
followers, 16, 86
  `commitIndex` updates by, 140
  committing, 29–33
  disconnected, 187
  handling read requests in, 340
  missing log entries in, 109
  reading from, 183, 188, 192
  replicating:
    log entries, 135–137
    updates to, 40–41
  serializing updates from the leader, 395

with the least network latency, 187
See also Leader and Followers pattern
Fowler, Martin, 13, 117

G

garbage collection, 9, 96, 103, 105–106
GCP (Google Cloud Platform), 3
Generation Clock pattern, 21–26, 103–107
as a Lamport clock, 307
detecting staled leaders with, 96, 141
for log entries, 111, 115, 136
for node metadata, 372
for partitions, 253
in Paxos, 117–119
in Raft, 141
leader elections in, 86–88, 133–134, 145–146
Go programming language
channels and goroutines in, 34, 159, 164–166
clones in, 200
lightweight threads in, 164
memberlist library of, 97
Google
Chubby, 339, 342, 354
data centers, 7–8
GCP, 3
Guava, 361
Percolator, 286–287
Spanner, 58, 128, 134, 261, 264, 291, 332
TrueTime, 10, 58, 299, 322, 325, 332
gossip convergence, 381
Gossip Dissemination pattern, 62–68, 363–373
eventual consistency and, 68, 372
failure detection in, 97
node selection in, 371
restarts in, 372
state exchanges in, 199, 368–372
tracking updates in, 381
vs. Consistent Core, 68, 372
gossip fanout, 366
GPS (Global Positioning System), 299, 325
Gray, Jim, 100
group membership, 339
Guava library (Google), 361
Gunther, Neil, 100

H

hardware, physical limits of, 3–6, 42
hash functions, 221, 229
HashiCorp
Consul, 96–98, 153, 157–158, 372–373
Serf Convergence Simulator, 364
`HashMap` class (Java), 74
Hazelcast platform
coordinators in, 222, 385, 392
executing client requests in, 388
failure detection in, 386
partitioning in, 221, 225, 241
HBase database (Apache)
idempotency in, 181
partitioning in, 44, 250, 255
HDFS (Hadoop Distributed File System), 343
head-of-line blocking, 96, 397
HeartBeat pattern, 17–19, 29, 32, 93–98
all-to-all, 386
detecting failures with, 222
in cluster members, 378, 385
in coordinators, 292
in leaders, 86, 88, 153, 157
expected from leaders, 95, 141–142
expiring sessions with, 180
including generations in, 105
network round trip time in, 94, 349
propagating the high-water mark with, 112
sending asynchronously in, 96
time intervals in, 94
tracking clusters in, 59
heterogeneous systems, 297
hierarchical storage, 339, 359
High-Water Mark pattern, 109–116
`commitIndex` as an example of, 115, 140
committing log entries in, 29–33
conflicting entries in, 104
generations in, 107
known by follower brokers, 192
leases and, 348, 350
majority of servers in, 100
propagating to followers, 112
read-your-writes consistency in, 41, 189
tracking replication with, 339
updating in, 34, 193, 229
Howard, Heidi, 132
Hybrid Clock pattern, 309–316
in Versioned Value, 318, 322
read-your-writes consistency in, 41, 189
tracking request order in, 54–58
used in two-phase commits, 279
vs. Lamport Clock, 310
with distributed transactions, 314–316
Hyperledger Fabric software, 134, 158, 373

I

idempotency, 35–36, 175–181
in key-value stores, 177, 273
idempotent producers (Kafka), 181
Idempotent Receiver pattern, 175–181

detecting duplicate requests with, 151, 342
request batching in, 404
updating data in, 35–36
Ignite database management system (Apache)
coordinators in, 392
failure detection in, 386
mapping in, 225
partitioning in, 43, 221, 241
IllegalStateException (Java), 165
Intel Optane memory, 74
Internet, time sources on, 301
isolation, 75

J

J2EE servers, 297
Java programming language
Date-Time API of, 325, 346
garbage collection in, 9
hash values in, 221
MVCC in, 194
Singular Update Queue implementation in, 160
java.lang.Thread class, 160–161
JDK (Java Development Kit), collection library of, 164
JGroups toolkit, 11
discovery protocols in, 376, 392
JMS (Java Message Service), 297
join requests, 377

K

Kafka platform (Apache), 11
blockchains and, 158
consistency in, 46
Controller Quorum in, 360
controllers in, 92, 166, 222
data clusters in, 58–59, 61
epoch in, 107
High-Water Mark in, 116
idempotency in, 181
logical storage structures in, 225
logs in:
cleaning, 83
segmentation of, 79
mapping data in, 220
messages from follower brokers in, 192
metadata in, 229, 343, 362
offsets in, 187
partitioning in, 241, 297, 340
pending requests in, 173
pull-based replication in, 151
request batching in, 404
request pipelines in, 408
single-socket channel in, 39, 398
storage implementation in, 76
timeouts in, 349, 354
transactions in, 297
Key-Range Partitions pattern, 44, 243–255
auto-splitting in, 249–255
load-based splitting in, 255
predefined key ranges in, 244–246
keys
all versions for, 197–199
mapped to partitions, 44, 221
ordering, 194–196, 200, 313
prefixes of, 339, 359
ranges of, 44, 241, 243–255
updating, 209
concurrently, 203–216
key-value records
adding, 75
setting, 151
key-value stores, 42, 127–130
changing keys in, 355
client interface of, 229, 245
current state of, 361
Hybrid Clock in, 322
key ranges in, 243–245
leases in, 351–352
mapping in, 219–220
Replicated Log in, 151
requests in:
idempotent, 177, 273
not ordered, 151
queued, 399–403
read, 152, 190
two-phase commits in, 258–260
updates in, 270
version numbers in, 193, 303–306, 312, 317
version vectors in, 207–210
Kotlin programming language, 164
Kubernetes system
coordinators in, 222
data clusters in, 58, 61
metadata in, 343
watch channels in, 362

L

Lamport Clock pattern, 104, 301–307
causal consistency in, 188
partially ordered values in, 54, 305, 318
tracking request order in, 52–54
used in two-phase commits, 279–280, 287
vs. Hybrid Clock, 310
Lamport timestamps. *See* timestamps, logical

Lamport, Leslie, 117, 302
latency, 186–187
Leader and Followers pattern, 16, 85–92
  consensus among nodes in, 117, 339
  coordinators in, 222
  generations of, 104–106
  majority of servers in, 100
  messages between, 395
  overloading with requests in, 183
  replicating:
    leases in, 345
    log entries in, 109, 193
  temporarily disconnected, 103
  using Generation Clock with, 21
  version numbers in, 305
  vs. Emergent Leader, 375, 392
leader brokers, 92
Leader Epoch (Kafka), 107
leader leases, 154, 157–158, 340
`leaderLeaseTimeout` interval, 155–157
leaderless replication, 208
leaders, 16, 85
  bottlenecks on, 341
  coordinating replication, 134
  deposed, 105
  electing, 17–27, 85–91, 96, 100, 104, 112, 134, 141–142, 340, 353
    continuously triggered, 338
    waiting prior to, 141
  executing log entries, order of, 140
  failures of, 17–26, 35, 59, 86, 91, 96, 109, 153, 157, 177, 342, 353
  finding, 151, 340–341, 392
  forwarding requests to, 340
  handling updates, 16
  maintaining:
    Lamport clock, 305–306
    leases, 350
    timeouts, 155, 157
  overloaded, 183
  reducing load on, 40
  responsive, 34–39
  sending:
    heartbeats to followers, 95, 141–142, 153, 157
    replicated log entries, 135, 137
  stale, 96, 141, 152
Lease pattern, 345–354
  cluster managing in, 337
  detecting failures in, 224
  distributing tasks across servers in, 342
  in key-value stores, 351–352
  leader failures and, 353
  log entries in, 151
  non-idempotent requests in, 177
  system timestamps in, 49
  tracking clusters in, 58–59, 352
leases
  creating, 177
  duplicated, 349
  expiring, 59, 345–346, 350
  leader, 154, 157–158, 340
  refreshing, 353
  registering, 349
  renewing, 345
  replicated, 59
lightweight threads, 164–165
linearizability, 90, 337, 340
`LinkedBlockingDeque` class (Java), 164
liveness, 99, 127
LMAX Disruptor library, 164, 166
load-based splitting, 255
locks, 159, 199, 258, 261–267
  deadlock prevention in, 262–267
  holding, 275, 279–280, 283
  releasing, 267, 270–272
  transaction isolation and, 261–262
  two-phase, 262
  using pending transactions as, 293–295
log entries, 14–15
  appending, 111, 135
  committing, 26–33
  conflicting, 104, 112, 115, 150
  corrupted, 74
  counters for, 193
  discarding, 81–83
  duplicated, 74
  executing order of, 140
  flushing, 73–74
  identifiers for, 71
  missing, 109
  previous generation of, 142
  replicating, 20, 109–111, 150
  resolving conflicts in, 21, 24
  sending to followers, 135
  storing states of, 133
  updating, 71, 159
  visible to clients, 112
LogCabin algorithm
  cluster time in, 346
  heartbeats in, 96
  idempotency in, 181
  single-socket channel in, 398
logical partitions, 220
logical storage structures, 225
logical timestamps, 299–333

logs
cleaning, 71, 76–77, 81–83, 151
flushing, 404
replicating, 133–158
segmented, 77–79
single, 71–76
size of, 77, 83
storing generations in, 104
syncing, 139
truncating, 112–115
Low-Water Mark pattern, 81–83
logs in:
cleaning, 71, 76, 151
segmenting, 74
time-based, 83
LWW (Last Write Wins) conflict resolution, 212–213, 216

M

Majority Quorum pattern, 20, 24, 99–102
building consensus with, 133–134, 137, 150–153
cluster size vs. throughput in, 337
committing log entries in, 26–29
data partitioning and, 45
in Paxos, 120–127, 132
leader elections in, 88, 141–142, 392
not sufficient for data consistency, 85
processing requests in, 167
replicating log entries in, 109
timeouts and, 155
tracking:
clusters in, 58
replication with, 339
updating data in, 34, 117
`matchIndex` (Raft), 139
MD5 hash algorithm, 221
memberlist library (Go), 97
membership
changes in, 375
maintaining list of, 378
missing updates of, 384
updating, 377, 379–383
memory, 3–6
partition splitting and, 255
queues and, 165
message brokers
distributed, 297
failure assumptions in, 134
mapping data in, 220
messages
acknowledgment, 379
delivery delays of, 9
ordering, 10, 408
processing, 163
metadata, 62
caching, 229
changes in, 342
creating, 254–255
implementing storage for, 339
propagation of, 62, 364–371
refreshing, 229
token, 237–241
transmitted at regular intervals, 365
microservices, 297
Microsoft Azure platform, 3
Cosmos DB, 128, 134
MongoDB database, 11
consistency in, 46, 191
data clusters in, 58, 61
Hybrid Clock in, 58, 279, 290, 309, 314, 316
latencies in, 186
log entries in, 151
MVCC backend in, 201, 307, 316
partitioning in, 297
replicas in, 187
Replicated Log in, 151
transactions in, 297
Multi-Paxos protocol, 128
implementing Replicated Log, 134, 158
Multi-Raft algorithm, 291
Murmur hash algorithm, 221
MVCC (multiversion concurrency control), 194, 201
hybrid timestamps in, 316
implemented in databases, 307
transaction isolation and, 199

N

Nagel's algorithm, 404
Neo4j database management system, 191
networks, 3–5
bandwidth of, 3, 97, 393
delays in, 299
failures of, 364
latency of, 399
partitioning in, 102
round trip time of, 94, 154, 349
temporarily disrupted, 103
throughput of, 404
nodes
adding, 42, 44, 230–235, 239–241, 381
age of, 375
agreeing upon themselves, 117
available, 24, 372

communications between, 167, 203, 393–408
continuously reading new requests, 395
counters for, 203
crashing, 14, 17–20, 117, 134, 258–259, 378
ephemeral, 90
equal, 375
executing different requests, 15
failures in, 222–224, 257, 270–272, 352, 372
identifiers of, 265, 378
lagging behind, 150
location of, 184
majority of, 100, 388
mapping data to, 219–222
multiple, 15
ordering, 375
overwhelmed, 407
partitioning data across, 42–44, 220–241
reconnecting to, 389
restarting, 372
seed, 375–377
sending:
gossip messages, 62–67, 363, 371
requests, 171
state of, 133, 363, 372
updating, 117–132, 363, 372
concurrent, 214
waiting, 141
nosql databases
durability of, 76
log segmentation in, 79
NTP service, 10, 212–213, 299, 301, 346

O

OFFSET_NOT_AVAILABLE error (Kafka), 192
operations
atomic, 74
read-modify-write, 261
write, 58, 101, 191, 273

P

parallel processing, 10
partitioning, 6, 13, 42–44, 217–297
bottlenecks in, 229
linearizability and, 340
number of partitions in, 219–241
ordering requests in, 151
quorums and, 102
resilience of, 45
partitions, 42
atomically stored values in, 297
consistency across, 46–49
distributing data across, 42
evenly, 220
fixed, 43–44, 219–235, 241
key-range, 44, 243–255
logical, 42, 220, 244
managing data in, 58–61
moving, 44, 220
proportional to number of nodes, 236–241
replicating, 45, 291
size of, 250–255
splitting points of, 44, 244
updating status of, 229
patterns
definition of, 10
See also individual patterns by name
Paxos pattern, 117–132
consensus in, 337–338
failures in, 134
flexible, 132
handling multiple values, 131
liveness vs. safety in, 127
phases of, 117–118
quorums in, 102, 120–127, 132
single-decree, 127
vs. Replicated Log, 257
PBFT algorithm, 134
Pebble database, 200
peer-to-peer systems, 375
Percolator data store (Google), 286–287
performance
communicating between servers and, 405
data replication and, 39
disks and, 3
flushing and, 74
log file size and, 77
parallel processing and, 10
partitioning and, 45
transaction isolation and, 261–262
Phi Accrual failure detector, 97–98
pickRandomNode method, 371
PostgresSQL database, 285
primary (Viewstamped Replication algorithm), 92
processes
causally related actions in, 104
crashing, 9
pausing, 9
waiting for response, 9
promises (Paxos), 120
proposers (Paxos), 118–119
competing, 127
pull-based replication, 151
Pulsar platform (Apache), 11
purgatory data structure, 173

Q

quartz crystals, 299
queues, 34–39
  backpressure on, 165
  singular update, 159–166
quorums
  flexible, 101
  handling responses of, 170
  intersection of, 102
  majority, 100–102
  size of, 101–102

R

Raft algorithm, 27
  `commitIndex` in, 115
  blockchains and, 158
  cluster time in, 346
  conflicting entries in, 115
  consensus in, 132, 337–339
  Consistent Core in, 372
  failures in, 98, 134
  idempotency in, 181
  implementing Replicated Log, 134–158, 291
  leader elections in, 87, 92, 112
  logs in, 76
    segmentation of, 79
  performance of, 39
  quorums in, 102
  replication lagging in, 183
  request pipelines in, 408
  requests in:
    expired, 179
    pipelined, 408
    processing, 166
  sending heartbeats in, 95
  single-socket channel in, 398
  snapshot mechanisms in, 82–83
  state-per-cluster nodes in, 151
  terms in, 107, 141, 143
  version numbers in, 193
Reactive Streams API, 358
read requests, 152–158, 183
  blocking, 262
  bypassing logs for, 152, 191
  dedicated threads for, 395–397
  handled by followers, 40–41, 340
  linearizable, 191
  repairing, 213–214
read restarts, 58, 322–324, 333
`ReadRestartException` (Java), 324
read-wait operation, 330–331
read-your-writes consistency, 40, 189
relational databases, 100
Replicated Log pattern, 27, 133–158
  bypassing logs for read requests in, 152, 191
  consistency in, 46–49
  handled by a separate thread, 34, 41
  in metadata storage, 339
  key-value stores and, 151
  partitioning data in, 45, 222–223
  performance of, 39
  persistent partition tables in, 226
  quorums in, 102
  registering requests in, 36–39
  storing mapping in, 245
  timestamp oracle in, 287
  tracking clusters in, 58, 61
  transactions in, 291
  version counters in, 306
  vs. Paxos, 128, 257
  vs. Write-Ahead Log, 74, 152
replication, 9, 13, 45, 69–216
  conflicting entries in, 115
  full, 139–140
  lagging, 183, 191
  leaderless, 208
  of client requests, 135–140
  pull-based, 151
  state machine, 158, 193
Request Batch pattern, 399–404
  asynchronous communication in, 167
  retry-backoff policy in, 404
request intervals, 94
Request Pipeline pattern, 405–408
  asynchronous communication in, 167
  for connected clients, 355
  for head-of-line blocking, 397
  heartbeats in, 96
  in-flight requests in, 179, 407
  request batching in, 404
Request Waiting List pattern, 167–173
  adding callbacks to, 168
  updating data in, 34–35
requests
  accepting, 21
  batching, 399–404
  connection, 342
  duplicated, 151, 175, 180–181, 342
  executing in different nodes, 15
  expiring, 172, 179
  failures in, 408
  handling, 15
  idempotent, 177, 273
  in separate threads, 405–406
  in-flight, 179, 407
  join, 377

non-idempotent, 177
one-way, 379
order of, 52–58, 151–152
overloading, 183
pending, 173, 400
processing, 161–173, 179, 181
processing time of, 399
read. *See* read requests
rejecting, 16, 408
repeated, 35, 175
responding to, 175
retrying, 181, 408
stale, 26
stopping serving of, 187
tracking, 35
waiting, 35
write. *See* write requests
resilience, 45
ResultCollector object, 379–381
retry-backoff policy, 404
Riak database
concurrent updates in, 214
LWW conflict resolution in, 212–213, 216
version vectors in, 204, 216
ring buffer data structure, 164
RocksDB database, 75
arranging data in, 194
clones of, 200
locks in, 293
sequence number in, 49
rollbacks, 271–272, 292
RSocket protocol, 358
rumors, 62–63, 364

S

S3 (Simple Storage Service), 188
safety, 99, 127
scalability, 5
schedulers, 94
SEDA architecture, 166
seed nodes, 375–377
Segmented Log pattern, 77–79
reducing storage space in, 74
using Write-Ahead Log with, 71
Serf Convergence Simulator (HashiCorp), 364
serializability, 340
serializable isolation, 261–262, 285
servers
changes on, 355
comparing timestamps on, 301
connection requests to, 342
distributing tasks across, 339
failures of, 364
detecting, 93–98
recovering after, 109
tolerating, 101
generation of, 104
number of, in a cluster, 100–102
overwhelming, 355
rebooting, 104
registering clients with, 175–177, 355–360
sessions on, 176
states of, 86
up-to-date, 87
service calls, external, 166
sessions, 176
expired, 180, 351
shard allocation (Akka), 241
shards. *See* partitions
sibling explosion, 216
Single Writer Principle, 166
Single-Socket Channel pattern, 395–398
between leader and followers, 39
for connected clients, 342, 355
heartbeats in, 96
performance and, 405
tracking clusters in, 61
Singular Update Queue pattern, 34, 159–166
backpressure in, 165
channels in, 164
heartbeats in, 96
Java implementation of, 161
performance and, 405
task chaining in, 166
threads in, 164
updates in, 71, 86
snapshot isolation, 199, 283–290
timestamps in, 286–290
snapshot mechanisms, 82–83
software systems
dealing with hardware physical limits, 6
problems in, 9–10
Spanner service (Google), 58
commit-wait in, 332
Multi-Paxos in, 128, 134
serializable isolation in, 261
timestamps in, 291
transactions in, 264
Spark engine (Apache), 343
split brain problem, 68, 386
state machine replication, 158, 193
State Watch pattern, 355–362
all events from a specific version in, 197–199
node failures in, 352
notification in, 342
tracking clusters in, 58–60

storage
  causal consistency in, 188
  embedded engines for, 200
  hierarchical, 339, 359
  limitations of, 74
  multiversion, 362
  stability of, 74
  transactional, 74–75
suspicion numbers, 97
SWIM (Scalable Weakly-Consistent Infection-Style Process Group Membership Protocol), 98, 372–373
  with Lifeguard enhancement, 97
synchronization, 49
`System.currentMillis`, `System.nanoTime` methods (Java), 346

T

tasks
  chaining, 166
  distributing across servers, 339, 342
  scheduled, 180, 346, 365–366, 384
TCP protocol, 9, 366, 395
  Nagel's algorithm in, 404
terms (Raft), 141, 143
`Thread` class (Java), 160–161
threads
  blocking, 159
  communicating, 165
  concurrent, 199
  dedicated for read and write requests, 395–397
  execution, 161–162
  handling, 34–39
  lightweight, 164–165
  separate for read and write requests, 405–406
  single, 159
three-way handshake, 371
`tick` method, 302
TiDB database, 286–287
TiKV database
  partitioning in, 255
  transactional intents in, 293
Time Sync Service (AWS), 322, 325, 333
time-bound leases. *See* Lease pattern
timeouts, 94, 155–157
  expiring requests after, 172
  for leases, 349
  measuring, 346
timestamp oracle service, 287–290
timestamps
  closed, 192
  comparing, 301
  conflict resolution with, 212–213
  handling, 316
  hybrid, 311–314, 316
  logical, 299–333
  monotonic, 286–290
  of heartbeats, 95
  ordering, 290–291, 313
  system, 49–56, 299, 301, 318, 325
    reading, 314
  used in two-phase commits, 279
token metadata, 237–241
transactional intents, 293–295
transactional outbox, 297
transactions, 74–75
  committing, 270
  concurrent, 262, 279, 285, 287
  conflicting, 275–279
  deadlocks in, 262–263
  distributed, 314–316
  fraudulent, 134
  identifiers of, 258
  incomplete, 292
  isolation of, 199, 261–262, 283–290
  lightweight, 128, 132
  logging, 291
  monotonic, 290
  ordering, 264–266
  pending, using as locks, 293–295
  read-only, 279
  read-write, 279–280, 283
  recording states of, 292
  restarting, 264, 291
  retried, 278
  rolling back, 263, 271–272, 292
  serializable, 285
TrueTime service (Google), 10, 58, 322, 325
  clock bound in, 332
  clock skew in, 299
Two-Phase Commit pattern, 257–297
  consistency in, 46–49
  failure handling in, 291–292
  Hybrid Clock in, 290
  in heterogeneous systems, 297
  snapshot isolation in, 199
  transactional intents in, 293–295
  transactions in, 75, 258–297, 314–316
  updating data in, 117
two-phase locking, 262

U

UDP (User Datagram Protocol), 366
updates
  atomic, 75, 257, 297

competing, 15–17
concurrent, 214
logging, 292
lost, 285
missing, 384
sending to other members, 379–383
two-phase, 257–297
UTC (Coordinated Universal Time), 50

V

values
changing, 355
partially ordered, 305, 318
storing version numbers of, 193
Vector Clock algorithm, 204
vector stamps, 203
version numbers, 193, 303
Version Vector, 203–216
comparing vectors in, 205
dotted, 216
implementations of, 204
in key-value stores, 207–210
read repairs in, 213–214
resolving conflicts in, 211–213
size of, 216
vs. Vector Clock, 204
Versioned Value pattern, 193–201
all events from a specific version in, 197–199
consistency in, 40–41, 49
Hybrid Clock in, 318, 322
ordering keys in, 194–196, 200, 313
storing values with, 306
timestamp oracle in, 287
transaction isolation in, 199
used in two-phase commits, 279–282
version numbers in, 303, 369
Viewstamped Replication algorithm, 92
Multi-Paxos in, 158
Voldemort database, 212
version vectors in, 205, 208, 216
votes
majority of, 100, 145–146
rejecting, 145
requesting, 141

W

wait-die policy, 267
waiting lists
adding callbacks to, 168, 171
expiring requests, 172
handling responses, 170, 172
invoking callbacks, 168, 172
maintaining, 167–168
WAL. *See* Write-Ahead Log
watch events, 356–362
web APIs, 5
wound-wait policy, 266, 279
write requests, 183
atomic, 273
buffered, 287
dedicated threads for, 395–397
in-flight, 291
repairing replicas after, 213
returning bookmarks, 191
throughput of, 101
waiting in, 58, 291
write skew, 285
Write-Ahead Log (WAL) pattern, 14–15, 19–20, 71–76
consistency in, 47
leader elections in, 87
node states in, 133
recording transaction states in, 292
saving write requests in, 193
storing generations in, 104–105
updating data in, 34, 159
used to recover after crashes, 109, 258–259
vs. Replicated Log, 152
write-behind logging, 74

X

XA transactions, 297

Y

YugabyteDB database, 11
data clusters in, 58, 61, 222
Hybrid Clock in, 58, 316
leader leases in, 158, 340
metadata in, 229
partitioning in, 44, 250, 254–255
read restart in, 333
transactional intents in, 293

Z

Zab algorithm
consensus in, 337
failures in, 98
leader elections in, 87, 92, 112
logs in, 76
Multi-Paxos in, 158
processing requests in, 166
request pipelines in, 408
ZooKeeper server (Apache), 11
data clusters in, 59, 61, 100
ephemeral nodes in, 90, 351
epoch in, 107

events in, 361–362
filesystem-like interface in, 339
forwarding client requests in, 340
leader elections in, 87, 90–92
metadata in, 229, 343
processing requests in, 166
read requests in, 152–153
sending heartbeats in, 95
sessions in, 181, 349, 351, 354
single-socket channel in, 39, 397
snapshot mechanisms in, 82–83
zxid (ZooKeeper transaction id), 181