Second Edition

UNDERSTANDING DISTRIBUTED SYSTEMS

What every developer should know about large distributed applications

ROBERTO VITILLO

Understanding Distributed Systems

Version 2.0.0

Roberto Vitillo

March 2022

Contents

Copyright ix

About the author xi

Acknowledgements xiii

Preface xv

1 Introduction 1
- 1.1 Communication . . . 2
- 1.2 Coordination . . . 3
- 1.3 Scalability . . . 3
- 1.4 Resiliency . . . 5
- 1.5 Maintainability . . . 6
- 1.6 Anatomy of a distributed system . . . 7

I Communication 11

2 Reliable links 17
- 2.1 Reliability . . . 18
- 2.2 Connection lifecycle . . . 18
- 2.3 Flow control . . . 20
- 2.4 Congestion control . . . 21
- 2.5 Custom protocols . . . 23

3 Secure links 25
- 3.1 Encryption . . . 25
- 3.2 Authentication . . . 26
- 3.3 Integrity . . . 28
- 3.4 Handshake . . . 29

4 Discovery **31**

5 APIs **35**
5.1 HTTP . . . 37
5.2 Resources . . . 39
5.3 Request methods . . . 41
5.4 Response status codes . . . 42
5.5 OpenAPI . . . 43
5.6 Evolution . . . 45
5.7 Idempotency . . . 46

II Coordination **53**

6 System models **57**

7 Failure detection **61**

8 Time **63**
8.1 Physical clocks . . . 63
8.2 Logical clocks . . . 65
8.3 Vector clocks . . . 67

9 Leader election **71**
9.1 Raft leader election . . . 72
9.2 Practical considerations . . . 73

10 Replication **77**
10.1 State machine replication . . . 78
10.2 Consensus . . . 81
10.3 Consistency models . . . 83
10.4 Chain replication . . . 90

11 Coordination avoidance **95**
11.1 Broadcast protocols . . . 96
11.2 Conflict-free replicated data types . . . 98
11.3 Dynamo-style data stores . . . 103
11.4 The CALM theorem . . . 105
11.5 Causal consistency . . . 106
11.6 Practical considerations . . . 110

12 Transactions **111**
12.1 ACID . . . 112
12.2 Isolation . . . 113

12.3 Atomicity 119
12.4 NewSQL 122

13 Asynchronous transactions 127
13.1 Outbox pattern 128
13.2 Sagas 130
13.3 Isolation 133

III Scalability 137

14 HTTP caching 145
14.1 Reverse proxies 148

15 Content delivery networks 151
15.1 Overlay network 151
15.2 Caching 153

16 Partitioning 155
16.1 Range partitioning 157
16.2 Hash partitioning 158

17 File storage 163
17.1 Blob storage architecture 163

18 Network load balancing 169
18.1 DNS load balancing 174
18.2 Transport layer load balancing 175
18.3 Application layer load balancing 178

19 Data storage 181
19.1 Replication 181
19.2 Partitioning 184
19.3 NoSQL 185

20 Caching 191
20.1 Policies 192
20.2 Local cache 193
20.3 External cache 194

21 Microservices 197
21.1 Caveats 199
21.2 API gateway 202

22 Control planes and data planes **209**
22.1 Scale imbalance 211
22.2 Control theory 214

23 Messaging **217**
23.1 Guarantees 221
23.2 Exactly-once processing 223
23.3 Failures 224
23.4 Backlogs 224
23.5 Fault isolation 225

IV Resiliency 229

24 Common failure causes **233**
24.1 Hardware faults 233
24.2 Incorrect error handling 234
24.3 Configuration changes 234
24.4 Single points of failure 235
24.5 Network faults 236
24.6 Resource leaks 237
24.7 Load pressure 238
24.8 Cascading failures 238
24.9 Managing risk 240

25 Redundancy **243**
25.1 Correlation 244

26 Fault isolation **247**
26.1 Shuffle sharding 248
26.2 Cellular architecture 250

27 Downstream resiliency **253**
27.1 Timeout 253
27.2 Retry 255
27.3 Circuit breaker 258

28 Upstream resiliency **261**
28.1 Load shedding 261
28.2 Load leveling 262
28.3 Rate-limiting 263
28.4 Constant work 269

V Maintainability 275

29 Testing 279
29.1 Scope . . . 280
29.2 Size . . . 281
29.3 Practical considerations . . . 283
29.4 Formal verification . . . 285

30 Continuous delivery and deployment 289
30.1 Review and build . . . 290
30.2 Pre-production . . . 292
30.3 Production . . . 293
30.4 Rollbacks . . . 293

31 Monitoring 297
31.1 Metrics . . . 298
31.2 Service-level indicators . . . 301
31.3 Service-level objectives . . . 304
31.4 Alerts . . . 306
31.5 Dashboards . . . 308
31.6 Being on call . . . 312

32 Observability 315
32.1 Logs . . . 316
32.2 Traces . . . 319
32.3 Putting it all together . . . 321

33 Manageability 323

34 Final words 327

Copyright

Understanding Distributed Systems by Roberto Vitillo

Copyright © Roberto Vitillo. All rights reserved.

The book's diagrams have been created with Excalidraw.

While the author has used good faith efforts to ensure that the information and instructions in this work are accurate, the author disclaims all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. The use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

About the author

Authors generally write this page in the third person as if someone else is writing about them, but I like to do things a little bit differently.

I have over 10 years of experience in the tech industry as a software engineer, technical lead, and manager.

After getting my master's degree in computer science from the University of Pisa, I worked on scientific computing applications at the Berkeley Lab. The software I contributed is used to this day by the ATLAS experiment at the Large Hadron Collider.

Next, I worked at Mozilla, where I set the direction of the data platform from its very early days and built a large part of it, including the team.

In 2017, I joined Microsoft to work on an internal SaaS for telemetry. Since then, I have helped launch multiple public SaaS products, like Playfab and Customer Insights. The data ingestion platform I am responsible for is one of the largest in the world, ingesting millions of events per second from billions of devices worldwide.

Acknowledgements

I am very thankful for the colleagues and mentors who inspired me and believed in me over the years. Thanks to Chiara Roda, Andrea Dotti, Paolo Calafiura, Vladan Djeric, Mark Reid, Pawel Chodarcewicz, and Nuno Cerqueira.

Rachael Churchill, Doug Warren, Vamis Xhagjika, Alessio Placitelli, Stefania Vitillo, and Alberto Sottile provided invaluable feedback on early drafts. Without them, the book wouldn't be what it is today.

A very special thanks to the readers of the first edition who reached out with feedback and suggestions over the past year and helped me improve the book: Matthew Williams, Anath B Chatterjee, Ryan Carney, Marco Vocialta, Curtis Washington, David Golden, Radosław Rusiniak, Sankaranarayanan Viswanathan, Praveen Barli, Abhijit Sarkar, Maki Pavlidis, Venkata Srinivas Namburi, Andrew Myers, Aaron Michaels, Jack Henschel, Ben Gummer, Luca Colantonio, Kofi Sarfo, Mirko Schiavone, and Gaurav Narula.

I am also very thankful for all the readers who left reviews or reached out to me and let me know they found the book useful. They gave me the motivation and energy to write a second edition.

Finally, and above all, thanks to my family: Rachell and Leonardo. Without your unwavering support, this book wouldn't exist.

Preface

Learning to build distributed systems is hard, especially if they are large scale. It's not that there is a lack of information out there. You can find academic papers, engineering blogs, and even books on the subject. The problem is that the available information is spread out all over the place, and if you were to put it on a spectrum from theory to practice, you would find a lot of material at the two ends but not much in the middle.

That is why I decided to write a book that brings together the core theoretical and practical concepts of distributed systems so that you don't have to spend hours connecting the dots. This book will guide you through the fundamentals of large-scale distributed systems, with just enough details and external references to dive deeper. This is the guide I wished existed when I first started out.

If you are a developer working on the backend of web or mobile applications (or would like to be!), this book is for you. When building distributed applications, you need to be familiar with the network stack, data consistency models, scalability and reliability patterns, observability best practices, and much more. Although you can build applications without knowing much of that, you will end up spending hours debugging and re-architecting them, learning hard lessons that you could have acquired in a much faster and less painful way.

However, if you have several years of experience designing and building highly available and fault-tolerant applications that scale to millions of users, this book might not be for you. As an expert,

you are likely looking for depth rather than breadth, and this book focuses more on the latter since it would be impossible to cover the field otherwise.

The second edition is a complete rewrite of the previous edition. Every page of the first edition has been reviewed and where appropriate reworked, with new topics covered for the first time.

This book is available both in a physical and digital format. The digital version is updated occasionally, which is why the book has a version number. You can subscribe to receive updates from the book's landing page[1].

As no book is ever perfect, I'm always happy to receive feedback. So if you find an error, have an idea for improvement, or simply want to comment on something, always feel free to write me[2]. I love connecting with readers!

[1] https://understandingdistributed.systems/
[2] roberto@understandingdistributed.systems

Chapter 1

Introduction

> *"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."*
>
> – Leslie Lamport

Loosely speaking, a distributed system is a group of nodes that cooperate by exchanging messages over communication links to achieve some task. A node can generically refer to a physical machine, like a phone, or a software process, like a browser.

Why do we bother building distributed systems in the first place?

Some applications are inherently distributed. For example, the web is a distributed system you are very familiar with. You access it with a browser, which runs on your phone, tablet, desktop, or Xbox. Together with other billions of devices worldwide, it forms a distributed system.

Another reason for building distributed systems is that some applications require high availability and need to be resilient to single-node failures. For example, Dropbox replicates your data across multiple nodes so that the loss of a single one doesn't cause your data to be lost.

Some applications need to tackle workloads that are just too big to fit on a single node, no matter how powerful. For example, Google receives tens of thousands of search requests per second from all over the globe. There is no way a single node could handle that.

And finally, some applications have performance requirements that would be physically impossible to achieve with a single node. Netflix can seamlessly stream movies to your TV at high resolution because it has a data center close to you.

This book tackles the fundamental challenges that need to be solved to design, build, and operate distributed systems.

1.1 Communication

The first challenge derives from the need for nodes to communicate with each other over the network. For example, when your browser wants to load a website, it resolves the server's IP address from the URL and sends an HTTP request to it. In turn, the server returns a response with the page's content.

How are the request and response messages represented on the wire? What happens when there is a temporary network outage, or some faulty network switch flips a few bits in the messages? How does the server guarantee that no intermediary can snoop on the communication?

Although it would be convenient to assume that some networking library is going to abstract all communication concerns away, in practice, it's not that simple because abstractions leak[1], and you need to understand how the network stack works when that happens.

[1] "The Law of Leaky Abstractions," https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/

1.2 Coordination

Another hard challenge of building distributed systems is that some form of coordination is required to make individual nodes work in unison towards a shared objective. This is particularly challenging to do in the presence of failures. The "two generals" problem is a famous thought experiment that showcases this.

Suppose two generals (nodes), each commanding their own army, need to agree on a time to jointly attack a city. There is some distance between the armies (network), and the only way to communicate is via messengers, who can be captured by the enemy (network failure). Under these assumptions, is there a way for the generals to agree on a time?

Well, general 1 could send a message with a proposed time to general 2. But since the messenger could be captured, general 1 wouldn't know whether the message was actually delivered. You could argue that general 2 could send a messenger with a response to confirm it received the original message. However, just like before, general 2 wouldn't know whether the response was actually delivered and another confirmation would be required. As it turns out, no matter how many rounds of confirmation are made, neither general can be certain that the other army will attack the city at the same time. As you can see, this problem is much harder to solve than it originally appeared.

Because coordination is such a key topic, the second part of the book is dedicated to understanding the fundamental distributed algorithms used to implement it.

1.3 Scalability

The performance of an application represents how efficiently it can handle *load*. Intuitively, load is anything that consumes the system's resources such as CPU, memory, and network bandwidth. Since the nature of load depends on the application's use cases and architecture, there are different ways to measure it. For example,

the number of concurrent users or the ratio of writes to reads are different forms of load.

For the type of applications discussed in this book, performance is generally measured in terms of throughput and response time. *Throughput* is the number of requests processed per second by the application, while *response time* is the time elapsed in seconds between sending a request to the application and receiving a response.

As load increases, the application will eventually reach its *capacity*, i.e., the maximum load it can withstand, when a resource is exhausted. The performance either plateaus or worsens at that point, as shown in Figure 1.1. If the load on the system continues to grow, it will eventually hit a point where most operations fail or time out.

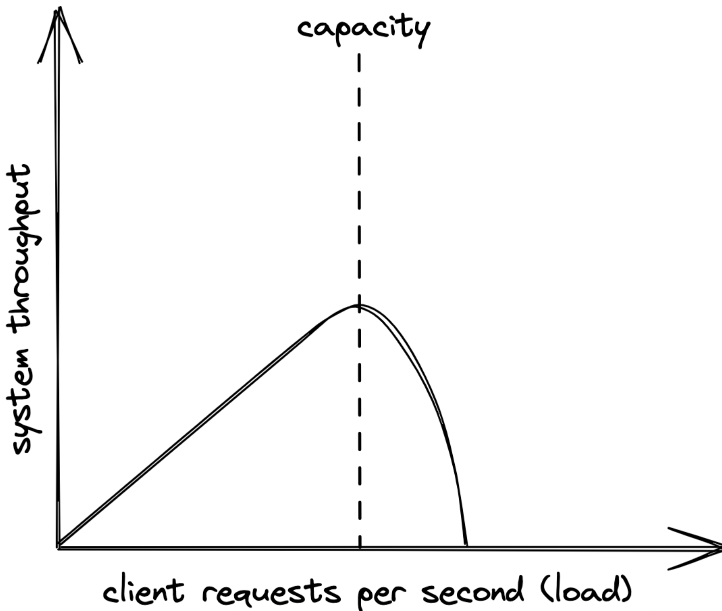

Figure 1.1: The system throughput on the y axis is the subset of client requests (x axis) that can be handled without errors and with low response times, also referred to as its goodput.

The capacity of a distributed system depends on its architecture, its implementation, and an intricate web of physical limitations like the nodes' memory size and clock cycle and the bandwidth and latency of network links. For an application to be scalable, a load increase should not degrade the application's performance. This requires increasing the capacity of the application at will.

A quick and easy way is to buy more expensive hardware with better performance, which is also referred to as *scaling up*. Unfortunately, this approach is bound to hit a brick wall sooner or later when such hardware just doesn't exist. The alternative is *scaling out* by adding more commodity machines to the system and having them work together.

Although procuring additional machines at will may have been daunting a few decades ago, the rise of cloud providers has made that trivial. In 2006 Amazon launched Amazon Web Services (AWS), which included the ability to rent virtual machines with its Elastic Compute Cloud (EC2[2]) service. Since then, the number of cloud providers and cloud services has only grown, democratizing the ability to create scalable applications.

In Part III of this book, we will explore the core architectural patterns and building blocks of scalable cloud-native applications.

1.4 Resiliency

A distributed system is resilient when it can continue to do its job even when failures happen. And at scale, anything that can go wrong will go wrong. Every component has a probability of failing — nodes can crash, network links can be severed, etc. No matter how small that probability is, the more components there are and the more operations the system performs, the higher the number of failures will be. And it gets worse because a failure of one component can increase the probability that another one will fail if the components are not well isolated.

[2]"Amazon EC2," https://aws.amazon.com/ec2/

Failures that are left unchecked can impact the system's *availability*[3], i.e., the percentage of time the system is available for use. It's a ratio defined as the amount of time the application can serve requests (*uptime*) divided by the total time measured (*uptime* plus *downtime*, i.e., the time the application can't serve requests).

Availability is often described with nines, a shorthand way of expressing percentages of availability. Three nines are typically considered acceptable by users, and anything above four is considered to be highly available.

Availability %	Downtime per day
90% ("one nine")	2.40 hours
99% ("two nines")	14.40 minutes
99.9% ("three nines")	1.44 minutes
99.99% ("four nines")	8.64 seconds
99.999% ("five nines")	864 milliseconds

If the system isn't resilient to failures, its availability will inevitably drop. Because of that, a distributed system needs to embrace failures and be prepared to withstand them using techniques such as redundancy, fault isolation, and self-healing mechanisms, which we will discuss in Part IV, *Resiliency*.

1.5 Maintainability

It's a well-known fact that the majority of the cost of software is spent after its initial development in maintenance activities, such as fixing bugs, adding new features, and operating it. Thus, we should aspire to make our systems easy to modify, extend and operate so that they are easy to maintain.

Any change is a potential incident waiting to happen. Good testing — in the form of unit, integration, and end-to-end tests — is a

[3] "AWS Well-Architected Framework, Availability," https://docs.aws.amazon.com/wellarchitected/latest/reliability-pillar/availability.html

minimum requirement to modify or extend a system without worrying it will break. And once a change has been merged into the codebase, it needs to be released to production safely without affecting the system's availability.

Also, operators need to monitor the system's health, investigate degradations and restore the service when it can't self-heal. This requires altering the system's behavior without code changes, e.g., toggling a feature flag or scaling out a service with a configuration change.

Historically, developers, testers, and operators were part of different teams, but the rise of microservices and DevOps has changed that. Nowadays, the same team that designs and implements a system is also responsible for testing and operating it. That's a good thing since there is no better way to discover where a system falls short than being on call for it. Part V will explore best practices for testing and operating distributed systems.

1.6 Anatomy of a distributed system

Distributed systems come in all shapes and sizes. In this book, we are mainly concerned with backend applications that run on commodity machines and implement some kind of business service. So you could say a distributed system is a group of machines that communicate over network links. However, from a run-time point of view, a distributed system is a group of software processes that communicate via *inter-process communication* (IPC) mechanisms like HTTP. And from an implementation perspective, a distributed system is a group of loosely-coupled components (services) that communicate via APIs. All these are valid and useful architectural points of view. In the rest of the book, we will switch between them depending on which one is more appropriate to discuss a particular topic.

A *service* implements one specific part of the overall system's capabilities. At the core of a service sits the business logic, which exposes interfaces to communicate with the outside world. Some

interfaces define the operations that the service offers to its users. In contrast, others define the operations that the service can invoke on other services, like data stores, message brokers, etc.

Since processes can't call each other's interfaces directly, *adapters* are needed to connect IPC mechanisms to service interfaces. An inbound adapter is part of the service's *Application Programming Interface* (API); it handles the requests received from an IPC mechanism, like HTTP, by invoking operations defined in the service interfaces. In contrast, outbound adapters grant the business logic access to external services, like data stores. This architectural style is also referred to as the ports and adapters architecture[4]. The idea is that the business logic doesn't depend on technical details; instead, the technical details depend on the business logic (dependency inversion principle[5]). This concept is illustrated in Figure 1.2.

Going forward, we will refer to a process running a service as a *server*, and a process sending requests to a server as a *client*. Sometimes, a process will be both a client and a server. For simplicity, we will assume that an individual instance of a service runs entirely within a single server process. Similarly, we will also assume that a process has a single thread. These assumptions will allow us to neglect some implementation details that would only complicate the discussion without adding much value.

[4] "Ports And Adapters Architecture," http://wiki.c2.com/?PortsAndAdaptersArchitecture

[5] "Dependency inversion principle," https://en.wikipedia.org/wiki/Dependency_inversion_principle

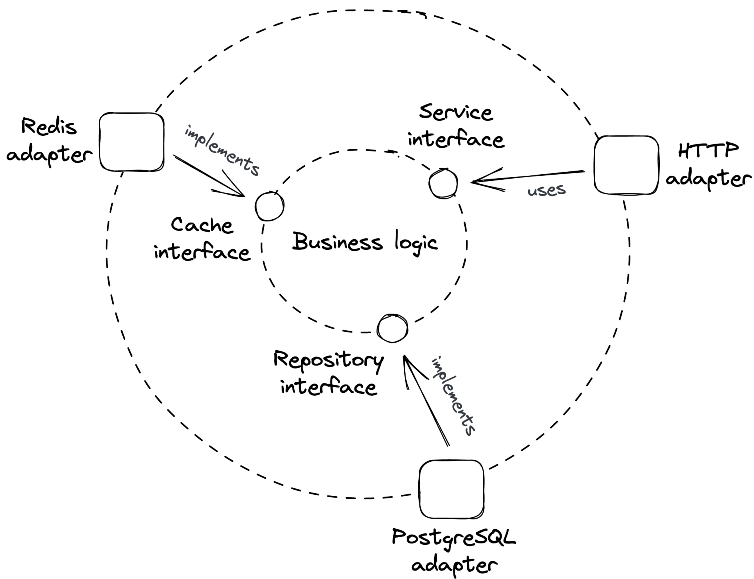

Figure 1.2: In this example, the business logic uses the repository interface, implemented by the PostgreSQL adapter, to access the database. In contrast, the HTTP adapter handles incoming requests by calling operations defined in the service interface.

Part I

Communication

Introduction

"The network is reliable."

– Fallacies of distributed computing, L. Peter Deutsch

Communication between processes over the network, or *inter-process communication* (IPC), is at the heart of distributed systems — it's what makes distributed systems distributed. In order for processes to communicate, they need to agree on a set of rules that determine how data is processed and formatted. Network protocols specify such rules.

The protocols are arranged in a stack[6], where each layer builds on the abstraction provided by the layer below, and lower layers are closer to the hardware. When a process sends data to another through the network stack, the data moves from the top layer to the bottom one and vice-versa at the other end, as shown in Figure 1.3:

- The *link layer* consists of network protocols that operate on local network links, like Ethernet or Wi-Fi, and provides an interface to the underlying network hardware. Switches operate at this layer and forward Ethernet packets based on their destination MAC address[7].
- The *internet layer* routes packets from one machine to another across the network. The Internet Protocol (IP) is the core protocol of this layer, which delivers packets on a best-effort ba-

[6] "Internet protocol suite," https://en.wikipedia.org/wiki/Internet_protocol_suite

[7] "MAC address," https://en.wikipedia.org/wiki/MAC_address

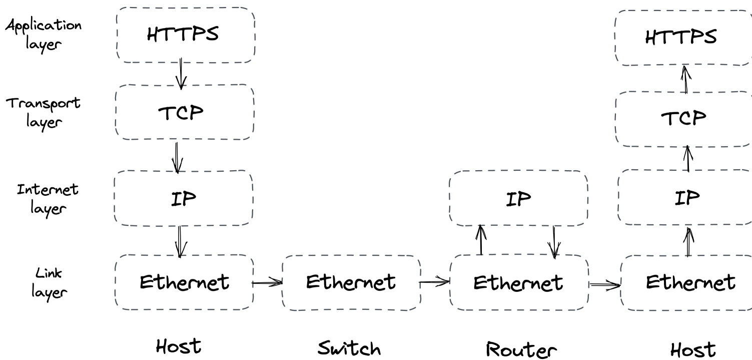

Figure 1.3: Internet protocol suite

sis (i.e., packets can be dropped, duplicated, or corrupted). Routers operate at this layer and forward IP packets to the next router along the path to their final destination.

- The *transport layer* transmits data between two processes. To enable multiple processes hosted on the same machine to communicate at the same time, port numbers are used to address the processes on either end. The most important protocol in this layer is the Transmission Control Protocol (TCP), which creates a reliable communication channel on top of IP.
- Finally, the *application layer* defines high-level communication protocols, like HTTP or DNS. Typically your applications will target this level of abstraction.

Even though each protocol builds on top of another, sometimes the abstractions leak. If you don't have a good grasp of how the lower layers work, you will have a hard time troubleshooting networking issues that will inevitably arise. More importantly, having an appreciation of the complexity of what happens when you make a network call will make you a better systems builder.

Chapter 2 describes how to build a reliable communication channel (TCP) on top of an unreliable one (IP), which can drop or duplicate data or deliver it out of order. Building reliable abstractions on top of unreliable ones is a common pattern we will encounter again in the rest of the book.

Chapter 3 describes how to build a secure channel (TLS) on top of a reliable one (TCP). Security is a core concern of any system, and in this chapter, we will get a taste of what it takes to secure a network connection from prying eyes and malicious agents.

Chapter 4 dives into how the phone book of the internet (DNS) works, which allows nodes to discover others using names. At its heart, DNS is a distributed, hierarchical, and eventually consistent key-value store. By studying it, we will get the first taste of eventual consistency[8] and the challenges it introduces.

Chapter 5 concludes this part by discussing how loosely coupled services communicate with each other through APIs by describing the implementation of a RESTful HTTP API built upon the protocols introduced earlier.

[8]We will learn more about consistency models in chapter 10.

Chapter 2

Reliable links

At the internet layer, the communication between two nodes happens by routing packets to their destination from one router to the next. Two ingredients are required for this: a way to address nodes and a mechanism to route packets across routers.

Addressing is handled by the IP protocol. For example, IPv6 provides a 128-bit address space, allowing 2^{128} addresses. To decide where to send a packet, a router needs to consult a local routing table. The table maps a destination address to the address of the next router along the path to that destination. The responsibility of building and communicating the routing tables across routers lies with the Border Gateway Protocol (BGP[1]).

Now, IP doesn't guarantee that data sent over the internet will arrive at its destination. For example, if a router becomes overloaded, it might start dropping packets. This is where TCP[2] comes in, a transport-layer protocol that exposes a reliable communication channel between two processes on top of IP. TCP guarantees that a stream of bytes arrives in order without gaps, duplication, or corruption. TCP also implements a set of stability patterns to

[1] "RFC 4271: A Border Gateway Protocol 4 (BGP-4)," https://datatracker.ietf.org/doc/html/rfc4271

[2] "RFC 793: Transmission Control Protocol," https://tools.ietf.org/html/rfc793

avoid overwhelming the network and the receiver.

2.1 Reliability

To create the illusion of a reliable channel, TCP partitions a byte stream into discrete packets called segments. The segments are sequentially numbered, which allows the receiver to detect holes and duplicates. Every segment sent needs to be acknowledged by the receiver. When that doesn't happen, a timer fires on the sending side and the segment is retransmitted. To ensure that the data hasn't been corrupted in transit, the receiver uses a checksum to verify the integrity of a delivered segment.

2.2 Connection lifecycle

A connection needs to be opened before any data can be transmitted on a TCP channel. The operating system manages the connection state on both ends through a *socket*. The socket keeps track of the state changes of the connection during its lifetime. At a high level, there are three states the connection can be in:

- The opening state in which the connection is being created.
- The established state in which the connection is open and data is being transferred.
- The closing state in which the connection is being closed.

In reality, this is a simplification, as there are more states[3] than the three above.

A server must be listening for connection requests from clients before a connection is established. TCP uses a three-way handshake to create a new connection, as shown in Figure 2.1:

1. The sender picks a random sequence number x and sends a SYN segment to the receiver.

[3] "TCP State Diagram," https://en.wikipedia.org/wiki/Transmission_Control_Protocol#/media/File:Tcp_state_diagram_fixed_new.svg

2. The receiver increments x, chooses a random sequence number y, and sends back a SYN/ACK segment.
3. The sender increments both sequence numbers and replies with an ACK segment and the first bytes of application data.

The sequence numbers are used by TCP to ensure the data is delivered in order and without holes.

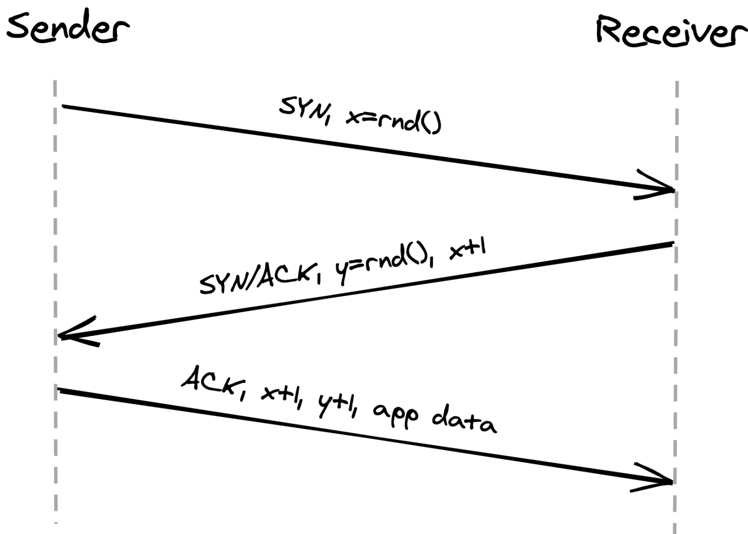

Figure 2.1: Three-way handshake

The handshake introduces a full round-trip in which no application data is sent. So until the connection has been opened, the bandwidth is essentially zero. The lower the round trip time is, the faster the connection can be established. Therefore, putting servers closer to the clients helps reduce this cold-start penalty.

After the data transmission is complete, the connection needs to be closed to release all resources on both ends. This termination phase involves multiple round-trips. If it's likely that another transmission will occur soon, it makes sense to keep the connection open to avoid paying the cold-start tax again.

Moreover, closing a socket doesn't dispose of it immediately as it

transitions to a waiting state (*TIME_WAIT*) that lasts several minutes and discards any segments received during the wait. The wait prevents delayed segments from a closed connection from being considered part of a new connection. But if many connections open and close quickly, the number of sockets in the waiting state will continue to increase until it reaches the maximum number of sockets that can be open, causing new connection attempts to fail. This is another reason why processes typically maintain connection pools to avoid recreating connections repeatedly.

2.3 Flow control

Flow control is a backoff mechanism that TCP implements to prevent the sender from overwhelming the receiver. The receiver stores incoming TCP segments waiting to be processed by the application into a receive buffer, as shown in Figure 2.2.

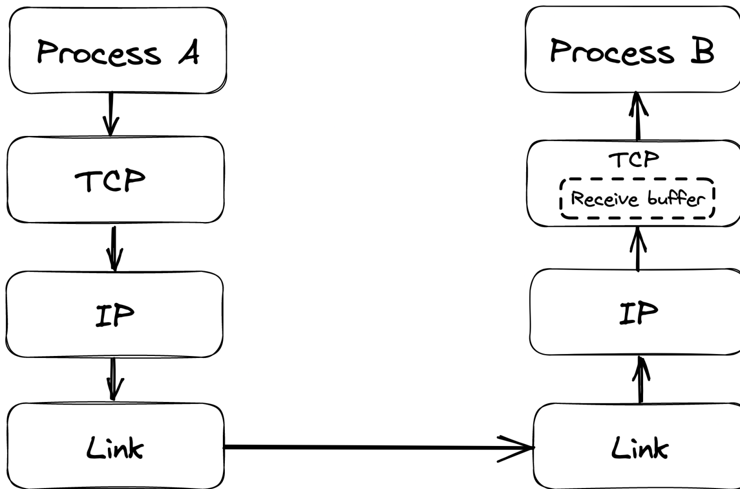

Figure 2.2: The receive buffer stores data that hasn't yet been processed by the destination process.

The receiver also communicates the size of the buffer to the sender whenever it acknowledges a segment, as shown in Figure 2.3.

Assuming it's respecting the protocol, the sender avoids sending more data than can fit in the receiver's buffer.

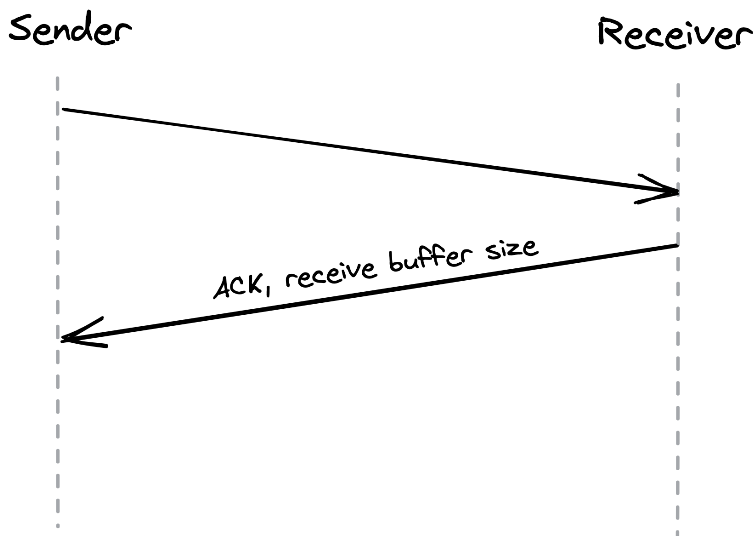

Figure 2.3: The size of the receive buffer is communicated in the headers of acknowledgment segments.

This mechanism is not too dissimilar to rate-limiting at the service level, a mechanism that rejects a request when a specific quota is exceeded (see section 28.3). But, rather than rate-limiting on an API key or IP address, TCP is rate-limiting on a connection level.

2.4 Congestion control

TCP guards not only against overwhelming the receiver, but also against flooding the underlying network. The sender maintains a so-called *congestion window*, which represents the total number of outstanding segments that can be sent without an acknowledgment from the other side. The smaller the congestion window is, the fewer bytes can be in flight at any given time, and the less bandwidth is utilized.

When a new connection is established, the size of the congestion window is set to a system default. Then, for every segment acknowledged, the window increases its size exponentially until it reaches an upper limit. This means we can't use the network's full capacity right after a connection is established. The shorter the round-trip time (RTT), the quicker the sender can start utilizing the underlying network's bandwidth, as shown in Figure 2.4.

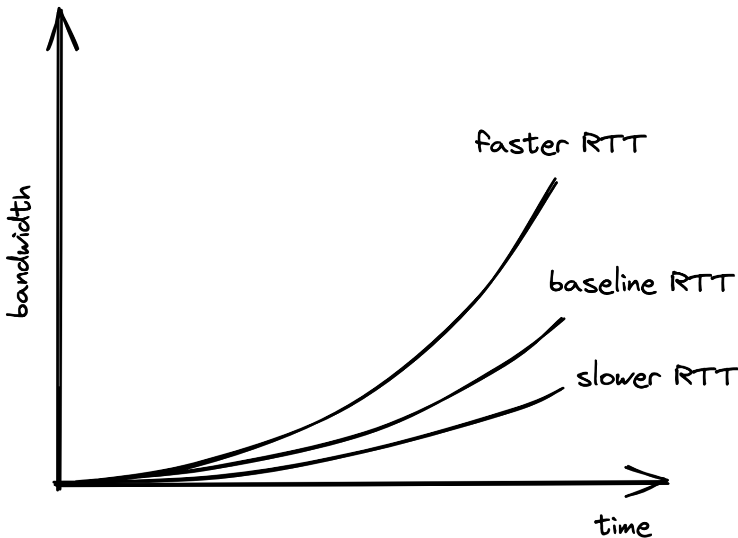

Figure 2.4: The shorter the RTT, the quicker the sender can start utilizing the underlying network's bandwidth.

What happens if a segment is lost? When the sender detects a missed acknowledgment through a timeout, a mechanism called *congestion avoidance* kicks in, and the congestion window size is reduced. From there onwards, the passing of time increases the window size[4] by a certain amount, and timeouts decrease it by another.

As mentioned earlier, the size of the congestion window defines the maximum number of bytes that can be sent without receiving

[4] "CUBIC: A New TCP-Friendly High-Speed TCP Variant," https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.153.3152&rep=rep1&type=pdf

an acknowledgment. Because the sender needs to wait for a full round trip to get an acknowledgment, we can derive the maximum theoretical bandwidth by dividing the size of the congestion window by the round trip time:

$$\text{Bandwidth} = \frac{\text{WinSize}}{\text{RTT}}$$

The equation[5] shows that bandwidth is a function of latency. TCP will try very hard to optimize the window size since it can't do anything about the round-trip time. However, that doesn't always yield the optimal configuration. Due to the way congestion control works, the shorter the round-trip time, the better the underlying network's bandwidth is utilized. This is more reason to put servers geographically close to the clients.

2.5 Custom protocols

TCP's reliability and stability come at the price of lower bandwidth and higher latencies than the underlying network can deliver. If we drop the stability and reliability mechanisms that TCP provides, what we get is a simple protocol named *User Datagram Protocol*[6] (UDP) — a connectionless transport layer protocol that can be used as an alternative to TCP.

Unlike TCP, UDP does not expose the abstraction of a byte stream to its clients. As a result, clients can only send discrete packets with a limited size called *datagrams*. UDP doesn't offer any reliability as datagrams don't have sequence numbers and are not acknowledged. UDP doesn't implement flow and congestion control either. Overall, UDP is a lean and bare-bones protocol. It's used to bootstrap custom protocols, which provide some, but not all, of the stability and reliability guarantees that TCP does[7].

[5] "Bandwidth-delay product," https://en.wikipedia.org/wiki/Bandwidth-delay_product

[6] "RFC 768: User Datagram Protocol," https://datatracker.ietf.org/doc/html/rfc768

[7] As we will later see, HTTP 3 is based on UDP to avoid some of TCP's shortcomings.

For example, in multiplayer games, clients sample gamepad events several times per second and send them to a server that keeps track of the global game state. Similarly, the server samples the game state several times per second and sends these snapshots back to the clients. If a snapshot is lost in transmission, there is no value in retransmitting it as the game evolves in real-time; by the time the retransmitted snapshot would get to the destination, it would be obsolete. This is a use case where UDP shines; in contrast, TCP would attempt to redeliver the missing data and degrade the game's experience.

Chapter 3

Secure links

We now know how to reliably send bytes from one process to another over the network. The problem is that these bytes are sent in the clear, and a middleman could intercept the communication. To protect against that, we can use the *Transport Layer Security*[1] (TLS) protocol. TLS runs on top of TCP and encrypts the communication channel so that application layer protocols, like HTTP, can leverage it to communicate securely. In a nutshell, TLS provides *encryption*, *authentication*, and *integrity*.

3.1 Encryption

Encryption guarantees that the data transmitted between a client and a server is obfuscated and can only be read by the communicating processes.

When the TLS connection is first opened, the client and the server negotiate a shared encryption secret using *asymmetric encryption*. First, each party generates a key pair consisting of a private and public key. The processes can then create a shared secret by exchanging their public keys. This is possible thanks to some math-

[1] "RFC 8446: The Transport Layer Security (TLS) Protocol Version 1.3," https://datatracker.ietf.org/doc/html/rfc8446

ematical properties[2] of the key pairs. The beauty of this approach is that the shared secret is never communicated over the wire.

Although asymmetric encryption is slow and expensive, it's only used to create the shared encryption key. After that, *symmetric encryption* is used, which is fast and cheap. The shared key is periodically renegotiated to minimize the amount of data that can be deciphered if the shared key is broken.

Encrypting in-flight data has a CPU penalty, but it's negligible since modern processors have dedicated cryptographic instructions. Therefore, TLS should be used for all communications, even those not going through the public internet.

3.2 Authentication

Although we have a way to obfuscate data transmitted across the wire, the client still needs to authenticate the server to verify it's who it claims to be. Similarly, the server might want to authenticate the identity of the client.

TLS implements authentication using digital signatures based on asymmetric cryptography. The server generates a key pair with a private and a public key and shares its public key with the client. When the server sends a message to the client, it signs it with its private key. The client uses the server's public key to verify that the digital signature was actually signed with the private key. This is possible thanks to mathematical properties[3] of the key pair.

The problem with this approach is that the client has no idea whether the public key shared by the server is authentic. Hence, the protocol uses certificates to prove the ownership of a public key. A certificate includes information about the owning entity, expiration date, public key, and a digital signature of the third-party entity that issued the certificate. The certificate's issuing

[2] "A (Relatively Easy To Understand) Primer on Elliptic Curve Cryptography," https://blog.cloudflare.com/a-relatively-easy-to-understand-primer-on-elliptic-curve-cryptography/

[3] "Digital signature," https://en.wikipedia.org/wiki/Digital_signature

entity is called a *certificate authority* (CA), which is also represented with a certificate. This creates a chain of certificates that ends with a certificate issued by a root CA, as shown in Figure 3.1, which self-signs its certificate.

For a TLS certificate to be trusted by a device, the certificate, or one of its ancestors, must be present in the trusted store of the client. Trusted root CAs, such as Let's Encrypt[4], are typically included in the client's trusted store by default by the operating system vendor.

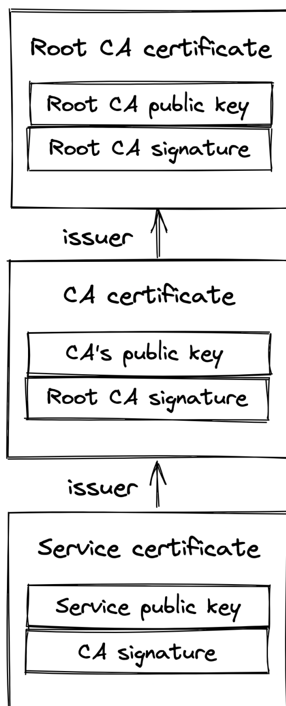

Figure 3.1: A certificate chain ends with a self-signed certificate issued by a root CA.

When a TLS connection is opened, the server sends the full certificate chain to the client, starting with the server's certificate and

[4] "Let's Encrypt: A nonprofit Certificate Authority," https://letsencrypt.org/

ending with the root CA. The client verifies the server's certificate by scanning the certificate chain until it finds a certificate that it trusts. Then, the certificates are verified in reverse order from that point in the chain. The verification checks several things, like the certificate's expiration date and whether the digital signature was actually signed by the issuing CA. If the verification reaches the last certificate in the path (the server's own certificate) without errors, the path is verified, and the server is authenticated.

One of the most common mistakes when using TLS is letting a certificate expire. When that happens, the client won't be able to verify the server's identity, and opening a connection to the remote process will fail. This can bring an entire application down as clients can no longer connect with it. For this reason, automation to monitor and auto-renew certificates close to expiration is well worth the investment.

3.3 Integrity

Even if the data is obfuscated, a middleman could still tamper with it; for example, random bits within the messages could be swapped. To protect against tampering, TLS verifies the integrity of the data by calculating a message digest. A secure hash function is used to create a message authentication code[5] (HMAC). When a process receives a message, it recomputes the digest of the message and checks whether it matches the digest included in the message. If not, then the message has either been corrupted during transmission or has been tampered with. In this case, the message is dropped.

The TLS HMAC protects against data corruption as well, not just tampering. You might be wondering how data can be corrupted if TCP is supposed to guarantee its integrity. While TCP does use a checksum to protect against data corruption, it's not 100% reliable[6]:

[5] "RFC 2104: HMAC: Keyed-Hashing for Message Authentication," https://datatracker.ietf.org/doc/html/rfc2104

[6] "When the CRC and TCP checksum disagree," https://dl.acm.org/doi/10.1145/347057.347561

it fails to detect errors for roughly 1 in 16 million to 10 billion packets. With packets of 1 KB, this is expected to happen once per 16 GB to 10 TB transmitted.

3.4 Handshake

When a new TLS connection is established, a handshake between the client and server occurs during which:

1. The parties agree on the cipher suite to use. A cipher suite specifies the different algorithms that the client and the server intend to use to create a secure channel, like the:
    - key exchange algorithm used to generate shared secrets;
    - signature algorithm used to sign certificates;
    - symmetric encryption algorithm used to encrypt the application data;
    - HMAC algorithm used to guarantee the integrity and authenticity of the application data.
2. The parties use the key exchange algorithm to create a shared secret. The symmetric encryption algorithm uses the shared secret to encrypt communication on the secure channel going forward.
3. The client verifies the certificate provided by the server. The verification process confirms that the server is who it says it is. If the verification is successful, the client can start sending encrypted application data to the server. The server can optionally also verify the client certificate if one is available.

These operations don't necessarily happen in this order, as modern implementations use several optimizations to reduce round trips. For example, the handshake typically requires 2 round trips with TLS 1.2 and just one with TLS 1.3. The bottom line is that creating a new connection is not free: yet another reason to put your servers geographically closer to the clients and reuse connections when possible.

Chapter 4

Discovery

So far, we have explored how to create a reliable and secure channel between two processes running on different machines. However, to create a new connection with a remote process, we must first discover its IP address somehow. The most common way of doing that is via the phone book of the internet: the *Domain Name System*[1] (DNS) — a distributed, hierarchical, and eventually consistent key-value store.

In this chapter, we will look at how DNS resolution[2] works in a browser, but the process is similar for other types of clients. When you enter a URL in your browser, the first step is to resolve the hostname's IP address, which is then used to open a new TLS connection. For example, let's take a look at how the DNS resolution works when you type *www.example.com* into your browser (see Figure 4.1).

1. The browser checks its local cache to see whether it has resolved the hostname before. If so, it returns the cached IP address; otherwise, it routes the request to a DNS resolver,

[1] "RFC 1035: Domain Names - Implementation and Specification," https://datatracker.ietf.org/doc/html/rfc1035

[2] "A deep dive into DNS," https://www.youtube.com/watch?v=drWd9HIhJdU

a server typically hosted by your Internet Service Provider (ISP).

2. The resolver is responsible for iteratively resolving the hostname for its clients. The reason why it's iterative will become obvious in a moment. The resolver first checks its local cache for a cached entry, and if one is found, it's returned to the client. If not, the query is sent to a root name server (root NS).

3. The root name server maps the *top-level domain* (TLD) of the request, i.e., *.com*, to the address of the name server responsible for it.

4. The resolver sends a resolution request for *example.com* to the TLD name server.

5. The TLD name server maps the *example.com* domain name to the address of the *authoritative name server* responsible for the domain.

6. Finally, the resolver queries the authoritative name server for *www.example.com*, which returns the IP address of the *www* hostname.

If the query included a subdomain of *example.com*, like *news.example.com*, the authoritative name server would have returned the address of the name server responsible for the subdomain, and an additional request would be required.

The original DNS protocol sent plain-text messages primarily over UDP for efficiency reasons. However, because this allows anyone monitoring the transmission to snoop, the industry has mostly moved to secure alternatives, such as DNS on top of TLS[3].

The resolution process involves several round trips in the worst case, but its beauty is that the address of a root name server is all that's needed to resolve a hostname. That said, the resolution would be slow if every request had to go through several name

[3]"RFC 7858: Specification for DNS over Transport Layer Security (TLS)," https://en.wikipedia.org/wiki/DNS_over_TLS

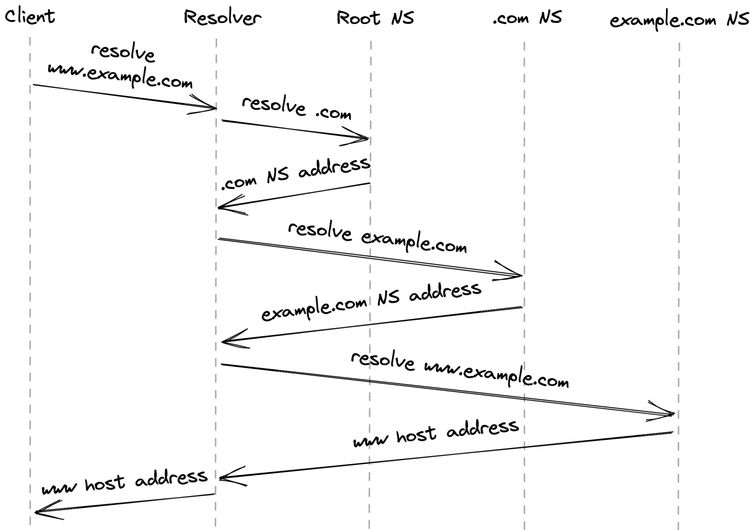

Figure 4.1: DNS resolution process

server lookups. Not only that, but think of the scale required for the name servers to handle the global resolution load. So caching is used to speed up the resolution process since the mapping of domain names to IP addresses doesn't change often — the browser, operating system, and DNS resolver all use caches internally.

How do these caches know when to expire a record? Every DNS record has a *time to live* (TTL) that informs the cache how long the entry is valid for. But there is no guarantee that clients play nicely and enforce the TTL. So don't be surprised when you change a DNS entry and find out that a small number of clients are still trying to connect to the old address long after the TTL has expired.

Setting a TTL requires making a tradeoff. If you use a long TTL, many clients won't see a change for a long time. But if you set it too short, you increase the load on the name servers and the average response time of requests because clients will have to resolve the hostname more often.

If your name server becomes unavailable for any reason, then the

smaller the record's TTL is, the higher the number of clients impacted will be. DNS can easily become a single point of failure — if your DNS name server is down and clients can't find the IP address of your application, they won't be able to connect it. This can lead to massive outages[4].

This brings us to an interesting observation. DNS could be a lot more robust to failures if DNS caches would serve stale entries when they can't reach a name server, rather than treating TTLs as time bombs. Since entries rarely change, serving a stale entry is arguably a lot more robust than not serving any entry at all. The principle that a system should continue to function even when a dependency is impaired is also referred to as "static stability"; we will talk more about it in the resiliency part of the book.

[4] "DDoS attack on Dyn," https://en.wikipedia.org/wiki/2016_Dyn_cyberattack

Chapter 5

APIs

Now that we know how a client can discover the IP address of a server and create a reliable and secure communication link with it, we want the client to invoke operations offered by the server. To that end, the server uses an adapter — which defines its application programming interface (API) — to translate messages received from the communication link to interface calls implemented by its business logic (see Figure 1.2).

The communication style between a client and a server can be *direct* or *indirect*, depending on whether the client communicates directly with the server or indirectly through a broker. Direct communication requires that both processes are up and running for the communication to succeed. However, sometimes this guarantee is either not needed or very hard to achieve, in which case indirect communication is a better fit. An example of indirect communication is *messaging*. In this model, the sender and the receiver don't communicate directly, but they exchange messages through a message channel (the broker). The sender sends messages to the channel, and on the other side, the receiver reads messages from it. Later in chapter 23, we will see how message channels are implemented and how to best use them.

In this chapter, we will focus our attention on a direct communi-

cation style called *request-response*, in which a client sends a *request message* to the server, and the server replies with a *response message*. This is similar to a function call but across process boundaries and over the network.

The request and response messages contain data that is serialized in a language-agnostic format. The choice of format determines a message's serialization and deserialization speed, whether it's human-readable, and how hard it is to evolve it over time. A *textual* format like JSON[1] is self-describing and human-readable, at the expense of increased verbosity and parsing overhead. On the other hand, a binary format like Protocol Buffers[2] is leaner and more performant than a textual one at the expense of human readability.

When a client sends a request to a server, it can block and wait for the response to arrive, making the communication *synchronous*. Alternatively, it can ask the outbound adapter to invoke a callback when it receives the response, making the communication *asynchronous*.

Synchronous communication is inefficient, as it blocks threads that could be used to do something else. Some languages, like JavaScript, C#, and Go, can completely hide callbacks through language primitives such as async/await[3]. These primitives make writing asynchronous code as straightforward as writing synchronous code.

Commonly used IPC technologies for request-response interactions are HTTP and gRPC[4]. Typically, internal APIs used for server-to-server communications within an organization are implemented with a high-performance RPC framework like gRPC. In contrast, external APIs available to the public tend to be based

[1] "ECMA-404: The JSON data interchange syntax," https://www.ecma-international.org/publications-and-standards/standards/ecma-404/

[2] "Protocol Buffers: a language-neutral, platform-neutral extensible mechanism for serializing structured data," https://developers.google.com/protocol-buffers

[3] "Asynchronous programming with async and await," https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/async/

[4] "gRPC: A high performance, open source universal RPC framework," https://grpc.io/

on HTTP, since web browsers can easily make HTTP requests via JavaScript code.

A popular set of design principles for designing elegant and scalable HTTP APIs is representational state transfer (REST[5]), and an API based on these principles is said to be RESTful. For example, these principles include that:

- requests are stateless, and therefore each request contains all the necessary information required to process it;
- responses are implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, the client can reuse the response for a later, equivalent request.

Given the ubiquity of RESTful HTTP APIs, we will walk through the process of creating an HTTP API in the rest of the chapter.

5.1 HTTP

HTTP[6] is a request-response protocol used to encode and transport information between a client and a server. In an *HTTP transaction*, the client sends a *request message* to the server's API endpoint, and the server replies back with a *response message*, as shown in Figure 5.1.

In HTTP 1.1, a message is a textual block of data that contains a start line, a set of headers, and an optional body:

- In a request message, the *start line* indicates what the request is for, and in a response message, it indicates whether the request was successful or not.
- The *headers* are key-value pairs with metadata that describes the message.
- The message *body* is a container for data.

HTTP is a stateless protocol, which means that everything needed by a server to process a request needs to be specified within the

[5] "Representational State Transfer," https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[6] "Hypertext Transfer Protocol," https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

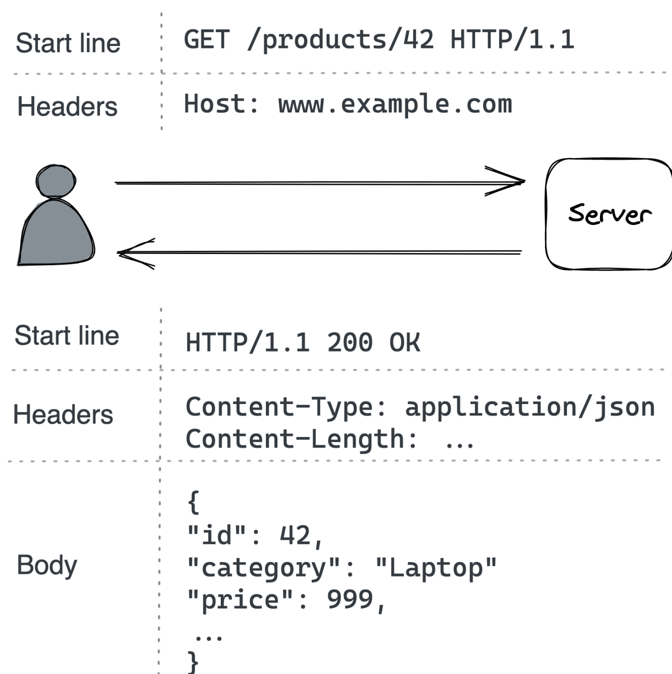

Figure 5.1: An example HTTP transaction between a browser and a web server

request itself, without context from previous requests. HTTP uses TCP for the reliability guarantees discussed in chapter 2. When it runs on top of TLS, it's also referred to as HTTPS.

HTTP 1.1 keeps a connection to a server open by default to avoid needing to create a new one for the next transaction. However, a new request can't be issued until the response to the previous one has been received (aka *head-of-line blocking* or HOL blocking); in other words, the transactions have to be serialized. For example, a browser that needs to fetch several images to render an HTML page has to download them one at a time, which can be very inefficient.

Although HTTP 1.1 technically allows some type of requests to be

pipelined[7], it still suffers from HOL blocking as a single slow response will block all the responses after it. With HTTP 1.1, the typical way to improve the throughput of outgoing requests is by creating multiple connections. However, this comes with a price because connections consume resources like memory and sockets.

HTTP 2[8] was designed from the ground up to address the main limitations of HTTP 1.1. It uses a binary protocol rather than a textual one, allowing it to multiplex multiple concurrent request-response transactions (streams) on the same connection. In early 2020 about half of the most-visited websites on the internet were using the new HTTP 2 standard.

HTTP 3[9] is the latest iteration of the HTTP standard, which is based on UDP and implements its own transport protocol to address some of TCP's shortcomings[10]. For example, with HTTP 2, a packet loss over the TCP connection blocks all streams (HOL), but with HTTP 3 a packet loss interrupts only one stream, not all of them.

This book uses the HTTP 1.1 standard for illustration purposes since its plain text format is easier to display. Moreover, HTTP 1.1 is still widely used.

5.2 Resources

Suppose we would like to implement a service to manage the product catalog of an e-commerce application. The service must allow customers to browse the catalog and administrators to create, update, or delete products. Although that sounds simple, in order to expose this service via HTTP, we first need to understand how to model APIs with HTTP.

[7] "HTTP pipelining," https://en.wikipedia.org/wiki/HTTP_pipelining

[8] "RFC 7540: Hypertext Transfer Protocol Version 2 (HTTP/2)," https://tools.ietf.org/html/rfc7540

[9] "HTTP/3 is next Generation HTTP. Is it QUIC enough?," https://www.youtube.com/watch?v=rlN4F1oyaRM

[10] "Comparing HTTP/3 vs. HTTP/2 Performance," https://blog.cloudflare.com/http-3-vs-http-2/

An HTTP server hosts resources, where a *resource* can be a physical or abstract entity, like a document, an image, or a collection of other resources. A URL identifies a resource by describing its location on the server.

In our catalog service, the collection of products is a type of resource, which could be accessed with a URL like *https://www.example.com/products?sort=price*, where:

- *https* is the protocol;
- *www.example.com* is the hostname;
- *products* is the name of the resource;
- *?sort=price* is the query string, which contains additional parameters that affect how the service handles the request; in this case, the sort order of the list of products returned in the response.

The URL without the query string is also referred to as the API's */products* endpoint.

URLs can also model relationships between resources. For example, since a product is a resource that belongs to the collection of products, the product with the unique identifier 42 could have the following relative URL: */products/42*. And if the product also has a list of reviews associated with it, we could append its resource name to the product's URL, i.e., */products/42/reviews*. However, the API becomes more complex as the nesting of resources increases, so it's a balancing act.

Naming resources is only one part of the equation; we also have to serialize the resources on the wire when they are transmitted in the body of request and response messages. When a client sends a request to get a resource, it adds specific headers to the message to describe the preferred representation for the resource. The server uses these headers to pick the most appropriate representation[11] for the response. Generally, in HTTP APIs, JSON is used to represent non-binary resources. For example, this is how the representation of */products/42* might look:

[11] "HTTP Content negotiation," https://developer.mozilla.org/en-US/docs/Web/HTTP/Content_negotiation

```
{
  "id": 42,
  "category": "Laptop",
  "price": 999
}
```

5.3 Request methods

HTTP requests can create, read, update, and delete (CRUD) resources using request *methods*. When a client makes a request to a server for a particular resource, it specifies which method to use. You can think of a request method as the verb or action to use on a resource.

The most commonly used methods are *POST*, *GET*, *PUT*, and *DELETE*. For example, the API of our catalog service could be defined as follows:

- *POST /products* — Create a new product and return the URL of the new resource.
- *GET /products* — Retrieve a list of products. The query string can be used to filter, paginate, and sort the collection.
- *GET /products/42* — Retrieve product 42.
- *PUT /products/42* — Update product 42.
- *DELETE /products/42* — Delete product 42.

Request methods can be categorized based on whether they are safe and whether they are idempotent. A *safe* method should not have any visible side effects and can safely be cached. An *idempotent* method can be executed multiple times, and the end result should be the same as if it was executed just a single time. Idempotency is a crucial aspect of APIs, and we will talk more about it later in section 5.7.

Method	Safe	Idempotent
POST	No	No
GET	Yes	Yes
PUT	No	Yes

Method	Safe	Idempotent
DELETE	No	Yes

5.4 Response status codes

After the server has received a request, it needs to process it and send a response back to the client. The HTTP response contains a *status code*[12] to communicate to the client whether the request succeeded or not. Different status code ranges have different meanings.

Status codes between 200 and 299 are used to communicate success. For example, *200 (OK)* means that the request succeeded, and the body of the response contains the requested resource.

Status codes between 300 and 399 are used for redirection. For example, *301 (Moved Permanently)* means that the requested resource has been moved to a different URL specified in the response message *Location* header.

Status codes between 400 and 499 are reserved for client errors. A request that fails with a client error will usually return the same error if it's retried since the error is caused by an issue with the client, not the server. Because of that, it shouldn't be retried. Some common client errors are:

- *400 (Bad Request)* — Validating the client-side input has failed.
- *401 (Unauthorized)* — The client isn't authenticated.
- *403 (Forbidden)* — The client is authenticated, but it's not allowed to access the resource.
- *404 (Not Found)* — The server couldn't find the requested resource.

Status codes between 500 and 599 are reserved for server errors. A request that fails with a server error can be retried as the issue that

[12] "HTTP Status Codes," https://httpstatuses.com/

caused it to fail might be temporary. These are some typical server status codes:

- *500 (Internal Server Error)* — The server encountered an unexpected error that prevented it from handling the request.
- *502 (Bad Gateway)* — The server, while acting as a gateway or proxy, received an invalid response from a downstream server it accessed while attempting to handle the request.[13]
- *503 (Service Unavailable)* — The server is currently unable to handle the request due to a temporary overload or scheduled maintenance.

5.5 OpenAPI

Now that we understand how to model an API with HTTP, we can write an adapter that handles HTTP requests by calling the business logic of the catalog service. For example, suppose the service is defined by the following interface:

```
interface CatalogService
{
  List<Product> GetProducts(...);
  Product GetProduct(...);
  void AddProduct(...);
  void DeleteProduct(...);
  void UpdateProduct(...)
}
```

So when the HTTP adapter receives a *GET /products* request to retrieve the list of all products, it will invoke the *GetProducts(...)* method and convert the result into an HTTP response. Although this is a simple example, you can see how the adapter connects the IPC mechanism (HTTP) to the business logic.

We can generate a skeleton of the HTTP adapter by defining the

[13] In this book, we will sometimes classify service dependencies as upstream or downstream depending on the direction of the dependency relationship. For example, if service A makes requests to service B, then service B is a downstream dependency of A, and A is an upstream dependency of B. Since there is no consensus in the industry for these terms, other texts might use a different convention.

API of the service with an *interface definition language* (IDL). An IDL is a language-independent definition of the API that can be used to generate boilerplate code for the server-side adapter and client-side software development kits (SDKs) in your languages of choice.

The OpenAPI[14] specification, which evolved from the Swagger project, is one of the most popular IDLs for RESTful HTTP APIs. With it, we can formally describe the API in a YAML document, including the available endpoints, supported request methods, and response status codes for each endpoint, and the schema of the resources' JSON representation.

For example, this is how part of the */products* endpoint of the catalog service's API could be defined:

```
openapi: 3.0.0
info:
  version: "1.0.0"
  title: Catalog Service API

paths:
  /products:
    get:
      summary: List products
      parameters:
        - in: query
          name: sort
          required: false
          schema:
            type: string
      responses:
        "200":
          description: list of products in catalog
          content:
            application/json:
              schema:
```

[14] "OpenAPI Specification," https://swagger.io/specification/

```
                type: array
                items:
                  $ref: "#/components/schemas/ProductItem"
        "400":
          description: bad input

components:
  schemas:
    ProductItem:
      type: object
      required:
        - id
        - name
        - category
      properties:
        id:
          type: number
        name:
          type: string
        category:
          type: string
```

Although this is a very simple example and we won't go deeper into OpenAPI, it should give you an idea of its expressiveness. With this definition, we can then run a tool to generate the API's documentation, boilerplate adapters, and client SDKs.

5.6 Evolution

An API starts out as a well-designed interface[15]. Slowly but surely, it will have to change to adapt to new use cases. The last thing we want to do when evolving an API is to introduce a breaking change that requires all clients to be modified at once, some of which we might have no control over.

There are two types of changes that can break compatibility, one

[15] Or at least it should.

at the endpoint level and another at the message level. For example, if we were to change the */products* endpoint to */new-products*, it would obviously break clients that haven't been updated to support the new endpoint. The same applies when making a previously optional query parameter mandatory.

Changing the schema of request or response messages in a backward-incompatible way can also wreak havoc. For example, changing the type of the *category* property in the *Product* schema from string to number is a breaking change that would cause the old deserialization logic to blow up in clients. Similar arguments[16] can be made for messages represented with other serialization formats, like Protocol Buffers.

REST APIs should be versioned to support breaking changes, e.g., by prefixing a version number in the URLs (*/v1/products/*). However, as a general rule of thumb, APIs should evolve in a backward-compatible way unless there is a very good reason. Although backward-compatible APIs tend not to be particularly elegant, they are practical.

5.7 Idempotency

When an API request times out, the client has no idea whether the server actually received the request or not. For example, the server could have processed the request and crashed right before sending a response back to the client.

An effective way for clients to deal with transient failures such as these is to retry the request one or more times until they get a response back. Some HTTP request methods (e.g., PUT, DELETE) are considered inherently idempotent as the effect of executing multiple identical requests is identical to executing only one request[17]. For example, if the server processes the same PUT request

[16] "Schema evolution in Avro, Protocol Buffers and Thrift," https://martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html

[17] "Idempotent Methods," https://datatracker.ietf.org/doc/html/rfc7231#section-4.2.2

for the same resource twice in a row, the end effect would be the same as if the PUT request was executed only once.

But what about requests that are not inherently idempotent? For example, suppose a client issues a POST request to add a new product to the catalog service. If the request times out, the client has no way of knowing whether the request succeeded or not. If the request succeeds, retrying it will create two identical products, which is not what the client intended.

In order to deal with this case, the client might have to implement some kind of reconciliation logic that checks for duplicate products and removes them when the request is retried. You can see how this introduces a lot of complexity for the client. Instead of pushing this complexity to the client, a better solution would be for the server to create the product only once by making the POST request idempotent, so that no matter how many times that specific request is retried, it will appear as if it only executed once.

For the server to detect that a request is a duplicate, it needs to be decorated with an idempotency key — a unique identifier (e.g., a UUID). The identifier could be part of a header, like *Idempotency-Key* in Stripe's API[18]. For the server to detect duplicates, it needs to remember all the request identifiers it has seen by storing them in a database. When a request comes in, the server checks the database to see if the request ID is already present. If it's not there, it adds the request identifier to the database and executes the request. Request identifiers don't have to be stored indefinitely, and they can be purged after some time.

Now here comes the tricky part. Suppose the server adds the request identifier to the database and crashes before executing the request. In that case, any future retry won't have any effect because the server will think it has already executed it. So what we really want is for the request to be handled atomically: either the server processes the request successfully and adds the request identifier to the database, or it fails to process it without storing the request

[18]"Designing robust and predictable APIs with idempotency," https://stripe.com/blog/idempotency

identifier.

If the request identifiers and the resources managed by the server are stored in the same database, we can guarantee atomicity with ACID transactions[19]. In other words, we can wrap the product creation and request identifier log within the same database transaction in the POST handler. However, if the handler needs to make external calls to other services to handle the request, the implementation becomes a lot more challenging[20], since it requires some form of coordination. Later, in chapter 13.2, we will learn how to do just that.

Now, assuming the server can detect a duplicate request, how should it be handled? In our example, the server could respond to the client with a status code that signals that the product already exists. But then the client would have to deal with this case differently than if it had received a successful response for the first POST request (*201 Created*). So ideally, the server should return the same response that it would have returned for the very first request.

So far, we have only considered a single client. Now, imagine the following scenario:

1. Client A sends a request to create a new product. Although the request succeeds, the client doesn't receive a timely response.
2. Client B deletes the newly created product.
3. Client A retries the original creation request.

How should the server deal with the request in step 3? From client's A perspective, it would be less surprising[21] to receive the original creation response than some strange error mentioning that the resource created with that specific request identifier has been deleted. When in doubt, it's helpful to follow the principle

[19] "Using Atomic Transactions to Power an Idempotent API," https://brandur.org/http-transactions
[20] "Implementing Stripe-like Idempotency Keys in Postgres," https://brandur.org/idempotency-keys
[21] "Making retries safe with idempotent APIs," https://aws.amazon.com/builders-library/making-retries-safe-with-idempotent-APIs/

of least astonishment.

To summarize, an idempotent API makes it a lot easier to implement clients that are robust to failures, since they can assume that requests can be retried on failure without worrying about all the possible edge cases.

Summary

Communicating over networks is what makes a system distributed. It's all too easy to ignore the "leaking" complexity that goes into it, since modern programming languages and frameworks make it look like invoking a remote API is no different from calling a local function. I know I did at first and eventually learned this lesson after spending days investigating weird degradations that ultimately turned out to be caused by exhausted socket pools or routers dropping packets.

Since then, I spend a great deal of time thinking of everything that can go wrong when a remote request is made: socket pools can be exhausted, TLS certificates can expire, DNS servers can become unavailable, routers can become congested and drop packets, non-idempotent requests can result in unexpected states, and the list goes on. The only universal truth is that the fastest, safest, most secure, and reliable network call is the one you don't have to make. And I hope that by reading about TCP, TLS, UDP, DNS, and HTTP, you also have (re)discovered a profound respect for the challenges of building networked applications.

Part II

Coordination

Introduction

> *"Here is a Raft joke. It is really a Paxos joke, but easier to follow."*
>
> – Aleksey Charapko

So far, we have learned how to get processes to communicate reliably and securely with each other. We didn't go into all this trouble just for the sake of it, though. Our ultimate goal is to build a distributed application made of a group of processes that gives its users the illusion they are interacting with one coherent node. Although achieving a perfect illusion is not always possible or desirable, some degree of coordination is always needed to build a distributed application.

In this part, we will explore the core distributed algorithms at the heart of distributed applications. This is the most challenging part of the book since it contains the most theory, but also the most rewarding one once you understand it. If something isn't clear during your first read, try to read it again and engage with the content by following the references in the text. Some of these references point to papers, but don't be intimidated. You will find that many papers are a lot easier to digest and more practical than you think. There is only so much depth a book can go into, and if you want to become an expert, reading papers should become a habit.

Chapter 6 introduces formal models that categorize systems based on what guarantees they offer about the behavior of processes, communication links, and timing; they allow us to reason about

distributed systems by abstracting away the implementation details.

Chapter 7 describes how to detect that a remote process is unreachable. Failure detection is a crucial component of any distributed system. Because networks are unreliable, and processes can crash at any time, without failure detection a process trying to communicate with another could hang forever.

Chapter 8 dives into the concept of time and order. We will first learn why agreeing on the time an event happened in a distributed system is much harder than it looks and then discuss a solution based on clocks that don't measure the passing of time.

Chapter 9 describes how a group of processes can elect a leader that can perform privileged operations, like accessing a shared resource or coordinating the actions of other processes.

Chapter 10 introduces one of the fundamental challenges in distributed systems: replicating data across multiple processes. This chapter also discusses the implications of the CAP and PACELC theorems, namely the tradeoff between consistency and availability/performance.

Chapter 11 explores weaker consistency models for replicated data, like strong eventual consistency and causal consistency, which allow us to build consistent, available, and partition-tolerant systems.

Chapter 12 dives into the implementation of ACID transactions that span data partitioned among multiple processes or services. Transactions relieve us from a whole range of possible failure scenarios so that we can focus on the actual application logic rather than all the possible things that can go wrong.

Chapter 13 discusses how to implement long-running atomic transactions that don't block by sacrificing the isolation guarantees of ACID. This chapter is particularly relevant in practice since it showcases techniques commonly used in microservice architectures.

Chapter 6

System models

To reason about distributed systems, we need to define precisely what can and can't happen. A *system model* encodes expectations about the behavior of processes, communication links, and timing; think of it as a set of assumptions that allow us to reason about distributed systems by ignoring the complexity of the actual technologies used to implement them.

For example, these are some common models for communication links:

- The *fair-loss link* model assumes that messages may be lost and duplicated, but if the sender keeps retransmitting a message, eventually it will be delivered to the destination.
- The *reliable link* model assumes that a message is delivered exactly once, without loss or duplication. A reliable link can be implemented on top of a fair-loss one by de-duplicating messages at the receiving side.
- The *authenticated reliable link* model makes the same assumptions as the reliable link but additionally assumes that the receiver can authenticate the sender.

Even though these models are just abstractions of real communication links, they are useful to verify the correctness of algorithms. And, as we have seen in the previous chapters, it's possible to build

a reliable and authenticated communication link on top of a fair-loss one. For example, TCP implements reliable transmission (and more), while TLS implements authentication (and more).

Similarly, we can model the behavior of processes based on the type of failures we expect to happen:

- The *arbitrary-fault* model assumes that a process can deviate from its algorithm in arbitrary ways, leading to crashes or unexpected behaviors caused by bugs or malicious activity. For historical reasons, this model is also referred to as the "Byzantine" model. More interestingly, it can be theoretically proven that a system using this model can tolerate up to $\frac{1}{3}$ of faulty processes[1] and still operate correctly.
- The *crash-recovery* model assumes that a process doesn't deviate from its algorithm but can crash and restart at any time, losing its in-memory state.
- The *crash-stop* model assumes that a process doesn't deviate from its algorithm but doesn't come back online if it crashes. Although this seems unrealistic for software crashes, it models unrecoverable hardware faults and generally makes the algorithms simpler.

The arbitrary-fault model is typically used to model safety-critical systems like airplane engines, nuclear power plants, and systems where a single entity doesn't fully control all the processes (e.g., digital cryptocurrencies such as Bitcoin). These use cases are outside the book's scope, and the algorithms presented here will generally assume a crash-recovery model.

Finally, we can also model timing assumptions:

- The *synchronous* model assumes that sending a message or executing an operation never takes more than a certain amount of time. This is not very realistic for the type of systems we care about, where we know that sending messages over the network can potentially take a very long

[1] "The Byzantine Generals Problem," https://lamport.azurewebsites.net/pubs/byz.pdf

time, and processes can be slowed down by, e.g., garbage collection cycles or page faults.
- The *asynchronous* model assumes that sending a message or executing an operation on a process can take an unbounded amount of time. Unfortunately, many problems can't be solved under this assumption; if sending messages can take an infinite amount of time, algorithms can get stuck and not make any progress at all. Nevertheless, this model is useful because it's simpler than models that make timing assumptions, and therefore algorithms based on it are also easier to implement[2].
- The *partially synchronous* model assumes that the system behaves synchronously most of the time. This model is typically representative enough of real-world systems.

In the rest of the book, we will generally assume a system model with fair-loss links, crash-recovery processes, and partial synchrony. If you are curious and want to learn more about other system models, "Introduction to Reliable and Secure Distributed Programming"[3] is an excellent theoretical book that explores distributed algorithms for a variety of models not considered in this text.

But remember, models are just an abstraction of reality[4] since they don't represent the real world with all its nuances. So, as you read along, question the models' assumptions and try to imagine how algorithms that rely on them could break in practice.

[2] "Unreliable Failure Detectors for Reliable Distributed Systems," https://www.cs.utexas.edu/~lorenzo/corsi/cs380d/papers/p225-chandra.pdf

[3] "Introduction to Reliable and Secure Distributed Programming," https://www.distributedprogramming.net/

[4] "All models are wrong," https://en.wikipedia.org/wiki/All_models_are_wrong

Chapter 7

Failure detection

Several things can go wrong when a client sends a request to a server. In the best case, the client sends a request and receives a response. But what if no response comes back after some time? In that case, it's impossible to tell whether the server is just very slow, it crashed, or a message couldn't be delivered because of a network issue (see Figure 7.1).

In the worst case, the client will wait forever for a response that will never arrive. The best it can do is make an educated guess, after some time has passed, on whether the server is unavailable. The client can configure a timeout to trigger if it hasn't received a response from the server after a certain amount of time. If and when the timeout triggers, the client considers the server unavailable and either throws an error or retries the request.

The tricky part is deciding how long to wait for the timeout to trigger. If the delay is too short, the client might wrongly assume the server is unavailable; if the delay is too long, the client might waste time waiting for a response that will never arrive. In summary, it's not possible to build a perfect failure detector.

But a process doesn't need to wait to send a message to find out that the destination is not reachable. It can also proactively try to maintain a list of available processes using pings or heartbeats.

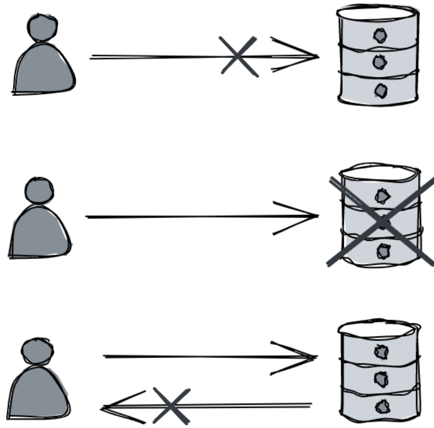

Figure 7.1: The client can't tell whether the server is slow, it crashed or a message was delayed/dropped because of a network issue.

A *ping* is a periodic request that a process sends to another to check whether it's still available. The process expects a response to the ping within a specific time frame. If no response is received, a timeout triggers and the destination is considered unavailable. However, the process will continue to send pings to it to detect if and when it comes back online.

A *heartbeat* is a message that a process periodically sends to another. If the destination doesn't receive a heartbeat within a specific time frame, it triggers a timeout and considers the process unavailable. But if the process comes back to life later and starts sending out heartbeats, it will eventually be considered to be available again.

Pings and heartbeats are generally used for processes that interact with each other frequently, in situations where an action needs to be taken as soon as one of them is no longer reachable. In other circumstances, detecting failures just at communication time is good enough.

Chapter 8

Time

Time is an essential concept in any software application; even more so in distributed ones. We have seen it play a crucial role in the network stack (e.g., DNS record TTL) and failure detection (timeouts). Another important use of it is for ordering events.

The flow of execution of a single-threaded application is simple to understand because every operation executes sequentially in time, one after the other. But in a distributed system, there is no shared global clock that all processes agree on that can be used to order operations. And, to make matters worse, processes can run concurrently.

It's challenging to build distributed applications that work as intended without knowing whether one operation happened before another. In this chapter, we will learn about a family of clocks that can be used to work out the order of operations across processes in a distributed system.

8.1 Physical clocks

A process has access to a physical wall-time clock. The most common type is based on a vibrating quartz crystal, which is cheap but not very accurate. Depending on manufacturing differences and

external temperature, one quartz clock can run slightly faster or slower than others. The rate at which a clock runs faster or slower is also called *clock drift*. In contrast, the difference between two clocks at a specific point in time is referred to as *clock skew*.

Because quartz clocks drift, they need to be synced periodically with machines that have access to higher-accuracy clocks, like atomic ones. Atomic clocks[1] measure time based on quantum-mechanical properties of atoms. They are significantly more expensive than quartz clocks and accurate to 1 second in 3 million years.

The synchronization between clocks can be implemented with a protocol, and the challenge is to do so despite the unpredictable latencies introduced by the network. The most commonly used protocol is the *Network Time Protocol* (NTP[2]). In NTP, a client estimates the clock skew by receiving a timestamp from a NTP server and correcting it with the estimated network latency. With an estimate of the clock skew, the client can adjust its clock. However, this causes the clock to jump forward or backward in time, which creates a problem when comparing timestamps. For example, an operation that runs after another could have an earlier timestamp because the clock jumped back in time between the two operations.

Luckily, most operating systems offer a different type of clock that is not affected by time jumps: a monotonic clock. A *monotonic clock* measures the number of seconds elapsed since an arbitrary point in time (e.g., boot time) and can only move forward. A monotonic clock is useful for measuring how much time has elapsed between two timestamps on the same node. However, monotonic clocks are of no use for comparing timestamps of different nodes.

Since we don't have a way to synchronize wall-time clocks across processes perfectly, we can't depend on them for ordering operations across nodes. To solve this problem, we need to look at it from another angle. We know that two operations can't run con-

[1] "Atomic clock," https://en.wikipedia.org/wiki/Atomic_clock

[2] "RFC 5905: Network Time Protocol Version 4: Protocol and Algorithms Specification," https://datatracker.ietf.org/doc/html/rfc5905

currently in a single-threaded process as one must happen before the other. This *happened-before* relationship creates a *causal* bond between the two operations, since the one that happens first can have side-effects that affect the operation that comes after it. We can use this intuition to build a different type of clock that isn't tied to the physical concept of time but rather captures the causal relationship between operations: a logical clock.

8.2 Logical clocks

A *logical clock* measures the passing of time in terms of logical operations, not wall-clock time. The simplest possible logical clock is a counter, incremented before an operation is executed. Doing so ensures that each operation has a distinct *logical timestamp*. If two operations execute on the same process, then necessarily one must come before the other, and their logical timestamps will reflect that. But what about operations executed on different processes?

Imagine sending an email to a friend. Any actions you did before sending that email, like drinking coffee, must have happened before the actions your friend took after receiving the email. Similarly, when one process sends a message to another, a so-called *synchronization point* is created. The operations executed by the sender before the message was sent *must* have happened before the operations that the receiver executed after receiving it.

A *Lamport clock*[3] is a logical clock based on this idea. To implement it, each process in the system needs to have a local counter that follows specific rules:

- The counter is initialized with 0.
- The process increments its counter by 1 before executing an operation.
- When the process sends a message, it increments its counter by 1 and sends a copy of it in the message.
- When the process receives a message, it merges the counter

[3] "Time, Clocks, and the Ordering of Events in a Distributed System," http://lamport.azurewebsites.net/pubs/time-clocks.pdf

it received with its local counter by taking the maximum of the two. Finally, it increments the counter by 1.

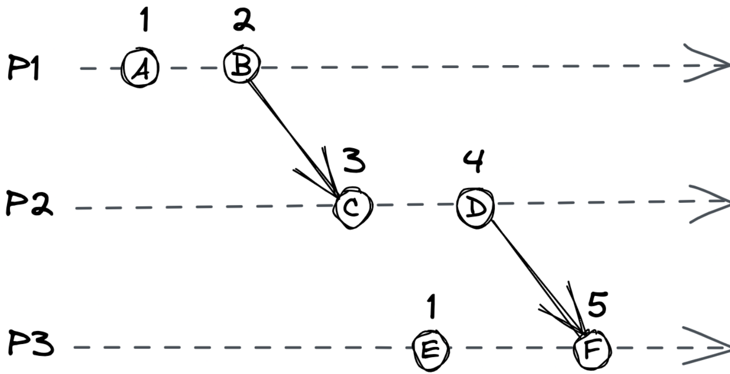

Figure 8.1: Three processes using Lamport clocks. For example, because D happened before F, D's logical timestamp is less than F's.

Although the Lamport clock assumes a crash-stop model, a crash-recovery one can be supported by e.g., persisting the clock's state on disk.

The rules guarantee that if operation O_1 happened-before operation O_2, the logical timestamp of O_1 is less than the one of O_2. In the example shown in Figure 8.1, operation D happened-before F, and their logical timestamps, 4 and 5, reflect that.

However, two unrelated operations can have the same logical timestamp. For example, the logical timestamps of operations A and E are equal to 1. To create a strict total order, we can arbitrarily order the processes to break ties. For example, if we used the process IDs in Figure 8.1 to break ties (1, 2, and 3), E's timestamp would be greater than A's.

Regardless of whether ties are broken, the order of logical timestamps doesn't imply a causal relationship. For example, in Figure 8.1, operation E didn't happen-before C, even if their timestamps appear to imply it. To guarantee this relationship, we have to use

a different type of logical clock: a *vector clock*.

8.3 Vector clocks

A *vector clock*[4] is a logical clock that guarantees that if a logical timestamp is less than another, then the former must have happened-before the latter. A vector clock is implemented with an array of counters, one for each process in the system. And, as with Lamport clocks, each process has its local copy.

For example, suppose the system is composed of three processes, $P1$, $P2$, and $P3$. In this case, each process has a local vector clock implemented with an array[5] of three counters $[C_{P1}, C_{P2}, C_{P3}]$. The first counter in the array is associated with $P1$, the second with $P2$, and the third with $P3$.

A process updates its local vector clock based on the following rules:

- Initially, the counters in the array are set to 0.
- When an operation occurs, the process increments its counter in the array by 1.
- When the process sends a message, it increments its counter in the array by 1 and sends a copy of the array with the message.
- When the process receives a message, it merges the array it received with the local one by taking the maximum of the two arrays element-wise. Finally, it increments its counter in the array by 1.

The beauty of vector clock timestamps is that they can be partially ordered[6]; given two operations O_1 and O_2 with timestamps T_1 and T_2, if:

[4] "Timestamps in Message-Passing Systems That Preserve the Partial Ordering," https://fileadmin.cs.lth.se/cs/Personal/Amr_Ergawy/dist-algos-papers/4.pdf

[5] In actual implementations, a dictionary is used rather than an array.

[6] In a total order, every pair of elements is comparable. Instead, in partial order, some pairs are not comparable

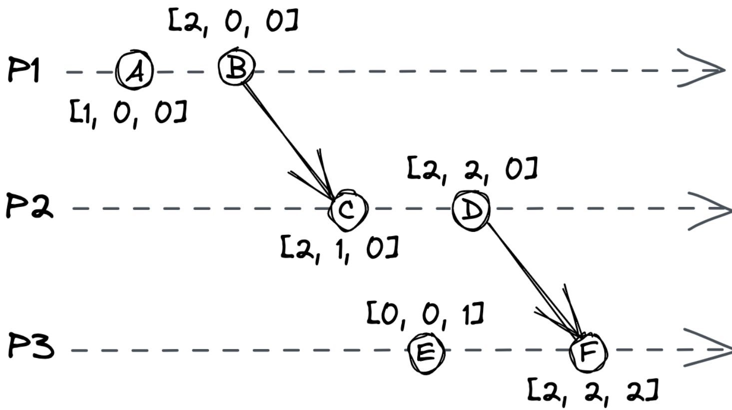

Figure 8.2: Each process has a vector clock represented by an array of three counters.

- every counter in T_1 is less than or equal to the corresponding counter in T_2,
- and there is at least one counter in T_1 that is strictly less than the corresponding counter in T_2,

then O_1 happened-before O_2. For example, in Figure 8.2, B happened-before C.

If O_1 didn't happen-before O_2 and O_2 didn't happen-before O_1, then the timestamps can't be ordered, and the operations are considered to be concurrent. So, for example, operations E and C in Figure 8.2 can't be ordered, and therefore are concurrent.

One problem with vector clocks is that the storage requirement on each process grows linearly with the number of processes, which becomes a problem for applications with many clients. However, there are other types of logical clocks that solve this issue, like dotted version vectors[7].

This discussion about logical clocks might feel a bit abstract at this point but bear with me. Later in the book, we will encounter some

[7] "Why Logical Clocks are Easy," https://queue.acm.org/detail.cfm?id=2917756

practical applications of logical clocks. What's important to internalize at this point is that, in general, we can't use physical clocks to accurately derive the order of events that happened on different processes. That being said, sometimes physical clocks are good enough. For example, using physical clocks to timestamp logs may be fine if they are only used for debugging purposes.

Chapter 9

Leader election

There are times when a single process in the system needs to have special powers, like accessing a shared resource or assigning work to others. To grant a process these powers, the system needs to elect a *leader* among a set of *candidate processes*, which remains in charge until it relinquishes its role or becomes otherwise unavailable. When that happens, the remaining processes can elect a new leader among themselves.

A leader election algorithm needs to guarantee that there is at most one leader at any given time and that an election eventually completes even in the presence of failures. These two properties are also referred to as *safety* and *liveness*, respectively, and they are general properties of distributed algorithms. Informally, safety guarantees that nothing bad happens and liveness that something good eventually does happen. In this chapter, we will explore how a specific algorithm, the *Raft* leader election algorithm, guarantees these properties.

9.1 Raft leader election

Raft[1]'s leader election algorithm is implemented as a state machine in which any process is in one of three states (see Figure 9.1):

- the *follower state*, where the process recognizes another one as the leader;
- the *candidate state*, where the process starts a new election proposing itself as a leader;
- or the *leader state*, where the process is the leader.

In Raft, time is divided into *election terms* of arbitrary length that are numbered with consecutive integers (i.e., logical timestamps). A term begins with a new election, during which one or more candidates attempt to become the leader. The algorithm guarantees that there is at most one leader for any term. But what triggers an election in the first place?

When the system starts up, all processes begin their journey as followers. A follower expects to receive a periodic heartbeat from the leader containing the election term the leader was elected in. If the follower doesn't receive a heartbeat within a certain period of time, a timeout fires and the leader is presumed dead. At that point, the follower starts a new election by incrementing the current term and transitioning to the candidate state. It then votes for itself and sends a request to all the processes in the system to vote for it, stamping the request with the current election term.

The process remains in the candidate state until one of three things happens: it wins the election, another process wins the election, or some time goes by with no winner:

- **The candidate wins the election** — The candidate wins the election if the majority of processes in the system vote for it. Each process can vote for at most one candidate in a term on a first-come-first-served basis. This majority rule enforces that at most one candidate can win a term. If the candidate wins the election, it transitions to the leader state and starts

[1] "In Search of an Understandable Consensus Algorithm," https://raft.github.io/raft.pdf

sending heartbeats to the other processes.

- **Another process wins the election** — If the candidate receives a heartbeat from a process that claims to be the leader with a term greater than or equal to the candidate's term, it accepts the new leader and returns to the follower state.[2] If not, it continues in the candidate state. You might be wondering how that could happen; for example, if the candidate process was to stop for any reason, like for a long garbage collection pause, by the time it resumes another process could have won the election.
- **A period of time goes by with no winner** — It's unlikely but possible that multiple followers become candidates simultaneously, and none manages to receive a majority of votes; this is referred to as a split vote. The candidate will eventually time out and start a new election when that happens. The election timeout is picked randomly from a fixed interval to reduce the likelihood of another split vote in the next election.

9.2 Practical considerations

There are other leader election algorithms out there, but Raft's implementation is simple to understand and also widely used in practice, which is why I chose it for this book. In practice, you will rarely, if ever, need to implement leader election from scratch. A good reason for doing that would be if you needed a solution with zero external dependencies[3]. Instead, you can use any *fault-tolerant* key-value store that offers a linearizable[4] *compare-and-swap*[5] operation with an expiration time (TTL).

The compare-and-swap operation atomically updates the value of a key if and only if the process attempting to update the value correctly identifies the current value. The operation takes three pa-

[2]The same happens if the leader receives a heartbeat with a greater term.

[3]We will encounter one such case when discussing replication in the next chapter.

[4]We will define what linearizability means exactly later in section 10.3.1.

[5]"Compare-and-swap," https://en.wikipedia.org/wiki/Compare-and-swap

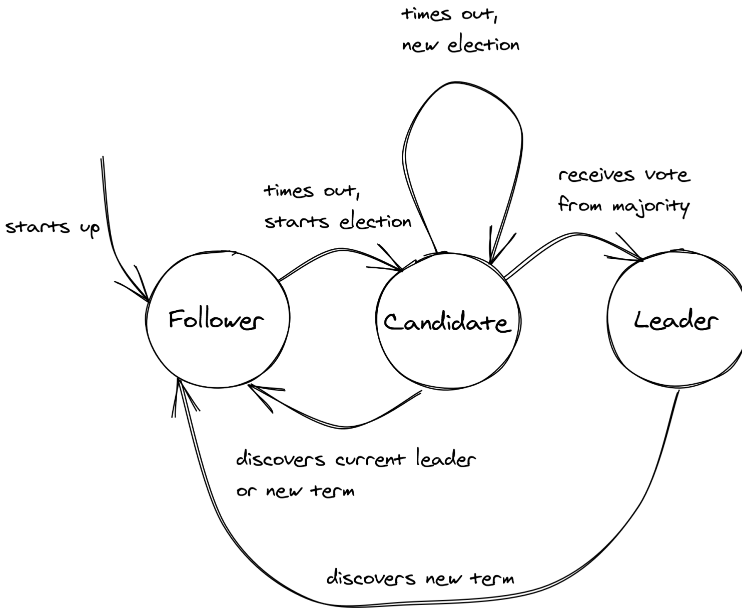

Figure 9.1: Raft's leader election algorithm represented as a state machine.

rameters: K, V_o, and V_n, where K is a key, and V_o and V_n are values referred to as the old and new value, respectively. The operation atomically compares the current value of K with V_o, and if they match, it updates the value of K to V_n. If the values don't match, then K is not modified, and the operation fails.

The expiration time defines the time to live for a key, after which the key expires and is removed from the store unless the expiration time is extended. The idea is that each competing process tries to acquire a *lease* by creating a new key with compare-and-swap. The first process to succeed becomes the leader and remains such until it stops renewing the lease, after which another process can become the leader.

The expiration logic can also be implemented on the client side,

like this locking library[6] for DynamoDB does, but the implementation is more complex, and it still requires the data store to offer a compare-and-swap operation.

You might think that's enough to guarantee there can't be more than one leader at any given time. But, unfortunately, that's not the case. To see why suppose multiple processes need to update a file on a shared file store, and we want to guarantee that only one at a time can access it to avoid race conditions. Now, suppose we use a lease to lock the critical section. Each process tries to acquire the lease, and the one that does so successfully reads the file, updates it in memory, and writes it back to the store:

```
if lease.acquire():
    try:
        content = store.read(filename)
        new_content = update(content)
        store.write(filename, new_content)
    except:
        lease.release()
```

The issue is that by the time the process gets to write to the file, it might no longer hold the lease. For example, the operating system might have preempted and stopped the process for long enough for the lease to expire. The process could try to detect that by comparing the lease expiration time to its local clock before writing to the store, assuming clocks are synchronized.

However, clock synchronization isn't perfectly accurate. On top of that, the lease could expire while the request to the store is in-flight because of a network delay. To account for these problems, the process could check that the lease expiration is far enough in the future before writing to the file. Unfortunately, this workaround isn't foolproof, and the lease can't guarantee mutual exclusion by itself.

To solve this problem, we can assign a version number to each file

[6] "Building Distributed Locks with the DynamoDB Lock Client," https://aws.amazon.com/blogs/database/building-distributed-locks-with-the-dynamodb-lock-client/

that is incremented every time the file is updated. The process holding the lease can then read the file and its version number from the file store, do some local computation, and finally update the file (and increment the version number) conditional on the version number not having changed. The process can perform this validation atomically using a compare-and-swap operation, which many file stores support.

If the file store doesn't support conditional writes, we have to design around the fact that occasionally there will be a race condition. Sometimes, that's acceptable; for example, if there are momentarily two leaders and they both perform the same idempotent update, no harm is done.

Although having a leader can simplify the design of a system as it eliminates concurrency, it can also become a scalability bottleneck if the number of operations performed by it increases to the point where it can no longer keep up. Also, a leader is a single point of failure with a large blast radius; if the election process stops working or the leader isn't working as expected, it can bring down the entire system with it. We can mitigate some of these downsides by introducing partitions and assigning a different leader per partition, but that comes with additional complexity. This is the solution many distributed data stores use since they need to use partitioning anyway to store data that doesn't fit in a single node.

As a rule of thumb, if we must have a leader, we have to minimize the work it performs and be prepared to occasionally have more than one.

Taking a step back, a crucial assumption we made earlier is that the data store that holds leases is fault-tolerant, i.e., it can tolerate the loss of a node. Otherwise, if the data store ran on a single node and that node were to fail, we wouldn't be able to acquire leases. For the data store to withstand a node failing, it needs to replicate its state over multiple nodes. In the next chapter, we will take a closer look at how this can be accomplished.

Chapter 10

Replication

Data replication is a fundamental building block of distributed systems. One reason for replicating data is to increase availability. If some data is stored exclusively on a single process, and that process goes down, the data won't be accessible anymore. However, if the data is replicated, clients can seamlessly switch to a copy. Another reason for replication is to increase scalability and performance; the more replicas there are, the more clients can access the data concurrently.

Implementing replication is challenging because it requires keeping replicas consistent with one another even in the face of failures. In this chapter, we will explore Raft's replication algorithm[1], a replication protocol that provides the strongest consistency guarantee possible — the guarantee that to the clients, the data appears to be stored on a single process, even if it's actually replicated. Arguably, the most popular protocol that offers this guarantee is Paxos[2], but we will discuss Raft as it's more understandable.

Raft is based on a mechanism known as *state machine replication*. The main idea is that a single process, the leader, *broadcasts* op-

[1] "In Search of an Understandable Consensus Algorithm," https://raft.github.io/raft.pdf

[2] "Paxos Made Simple," https://lamport.azurewebsites.net/pubs/paxos-simple.pdf

erations that change its state to other processes, the followers (or replicas). If the followers execute the same sequence of operations as the leader, then each follower will end up in the same state as the leader. Unfortunately, the leader can't simply broadcast operations to the followers and call it a day, as any process can fail at any time, and the network can lose messages. This is why a large part of the algorithm is dedicated to fault tolerance.

The reason why this this mechanism is called stated machine replication is that each process is modeled as a *state machine*[3] that transitions from one state to another in response to some input (an operation). If the state machines are *deterministic* and get exactly the same input in the same order, their states are consistent. That way, if one of them fails, a redundant copy is available from any of the other state machines. State machine replication is a very powerful tool to make a service fault-tolerant as long it can be modeled as a state machine.

For example, consider the problem of implementing a fault-tolerant key-value store. In this case, each state machine represents a storage node that accepts *put(k, v)* and *get(k)* operations. The actual state is represented with a dictionary. When a *put* operation is executed, the key-value pair is added to the dictionary. When a *get* operation is executed, the value corresponding to the requested key is returned. You can see how if every node executes the same sequence of *puts*, all nodes will end up having the same state.

In the next section, we will take a deeper look at Raft's replication protocol. It's a challenging read that requires to pause and think, but I can assure you it's well worth the effort, especially if you haven't seen a replication protocol before.

10.1 State machine replication

When the system starts up, a leader is elected using Raft's leader election algorithm discussed in chapter 9, which doesn't require

[3] "Finite-state machine," https://en.wikipedia.org/wiki/Finite-state_machine

any external dependencies. The leader is the only process that can change the replicated state. It does so by storing the sequence of operations that alter the state into a local *log*, which it replicates to the followers. Replicating the log is what allows the state to be kept in sync across processes.

As shown in Figure 10.1, a log is an ordered list of entries where each entry includes:

- the operation to be applied to the state, like the assignment of 3 to x. The operation needs to be deterministic so that all followers end up in the same state, but it can be arbitrarily complex as long as that requirement is respected (e.g., compare-and-swap or a transaction with multiple operations);
- the index of the entry's position in the log;
- and the leader's election term (the number in each box).

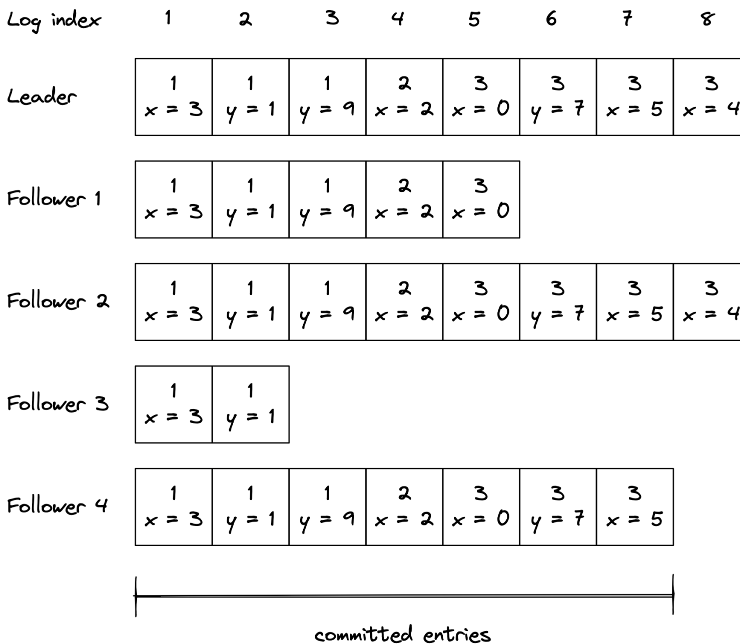

Figure 10.1: The leader's log is replicated to its followers.

When the leader wants to apply an operation to its local state, it

first appends a new entry for the operation to its log. At this point, the operation hasn't been applied to the local state just yet; it has only been logged.

The leader then sends an *AppendEntries* request to each follower with the new entry to be added. This message is also sent out periodically, even in the absence of new entries, as it acts as a *heartbeat* for the leader.

When a follower receives an *AppendEntries* request, it appends the entry it received to its own log (without actually executing the operation yet) and sends back a response to the leader to acknowledge that the request was successful. When the leader hears back successfully from a majority of followers, it considers the entry to be committed and executes the operation on its local state. The leader keeps track of the highest committed index in the log, which is sent in all future *AppendEntries* requests. A follower only applies a log entry to its local state when it finds out that the leader has committed the entry.

Because the leader needs to wait for *only* a majority (quorum) of followers, it can make progress even if some are down, i.e., if there are $2f + 1$ followers, the system can tolerate up to f failures. The algorithm guarantees that an entry that is committed is durable and will eventually be executed by all the processes in the system, not just those that were part of the original majority.

So far, we have assumed there are no failures, and the network is reliable. Let's relax those assumptions. If the leader fails, a follower is elected as the new leader. But, there is a caveat: because the replication algorithm only needs a majority of processes to make progress, it's possible that some processes are not up to date when a leader fails. To avoid an out-of-date process becoming the leader, a process can't vote for one with a less up-to-date log. In other words, a process can't win an election if it doesn't contain all committed entries.

To determine which of two processes' logs is more up-to-date, the election term and index of their last entries are compared. If the logs end with different terms, the log with the higher term is more

up to date. If the logs end with the same term, whichever log is longer is more up to date. Since the election requires a majority vote, and a candidate's log must be at least as up to date as any other process in that majority to win the election, the elected process will contain all committed entries.

If an *AppendEntries* request can't be delivered to one or more followers, the leader will retry sending it indefinitely until a majority of the followers have successfully appended it to their logs. Retries are harmless as *AppendEntries* requests are idempotent, and followers ignore log entries that have already been appended to their logs.

If a follower that was temporarily unavailable comes back online, it will eventually receive an *AppendEntries* message with a log entry from the leader. The *AppendEntries* message includes the index and term number of the entry in the log that immediately precedes the one to be appended. If the follower can't find a log entry with that index and term number, it rejects the message to prevent creating a gap in its log.

When the *AppendEntries* request is rejected, the leader retries the request, this time including the last two log entries — this is why we referred to the request as *AppendEntries* and not as *AppendEntry*. If that fails, the leader retries sending the last three log entries and so forth.[4] The goal is for the leader to find the latest log entry where the two logs agree, delete any entries in the follower's log after that point, and append to the follower's log all of the leader's entries after it.

10.2 Consensus

By solving state machine replication, we actually found a solution to *consensus*[5] — a fundamental problem studied in distributed systems research in which a group of processes has to decide a value

[4] In practice, there are ways to reduce the number of messages required for this step.

[5] "Consensus," https://en.wikipedia.org/wiki/Consensus_(computer_science)

so that:

- every non-faulty process eventually agrees on a value;
- the final decision of every non-faulty process is the same everywhere;
- and the value that has been agreed on has been proposed by a process.

This may sound a little bit abstract. Another way to think about consensus is as the API of a write-once register[6] (WOR): a thread-safe and linearizable[7] register that can only be written once but can be read many times.

There are plenty of practical applications of consensus. For example, agreeing on which process in a group can acquire a lease requires consensus. And, as mentioned earlier, state machine replication also requires it. If you squint a little, you should be able to see how the replicated log in Raft is a sequence of WORs, and so Raft really is just a sequence of consensus instances.

While it's important to understand what consensus is and how it can be solved, you will likely never need to implement it from scratch[8]. Instead, you can use one of the many off-the-shelf solutions available.

For example, one of the most common uses of consensus is for coordination purposes, like the election of a leader. As discussed in 9.2, leader election can be implemented by acquiring a lease. The lease ensures that at most one process can be the leader at any time and if the process dies, another one can take its place. However, this mechanism requires the lease manager, or coordination service, to be fault-tolerant. Etcd[9] and ZooKeeper[10] are two widely used co-

[6] "Paxos made Abstract," https://maheshba.bitbucket.io/blog/2021/11/15/Paxos.html

[7] We will define what linearizability means in the next section.

[8] nor want to, since it's very challenging to get right; see "Paxos Made Live - An Engineering Perspective," https://static.googleusercontent.com/media/research.google.com/en//archive/paxos_made_live.pdf

[9] "etcd: A distributed, reliable key-value store for the most critical data of a distributed system," https://etcd.io/

[10] "Apache ZooKeeper: An open-source server which enables highly reliable distributed coordination," https://zookeeper.apache.org/

ordination services that replicate their state for fault-tolerance using consensus. A coordination service exposes a hierarchical, key-value store through its API, and also allows clients to watch for changes to keys. So, for example, acquiring a lease can be implemented by having a client attempt to create a key with a specific TTL. If the key already exists, the operation fails guaranteeing that only one client can acquire the lease.

10.3 Consistency models

We discussed state machine replication with the goal of implementing a data store that can withstand failures and scale out to serve a larger number of requests. Now that we know how to build a replicated data store in principle, let's take a closer look at what happens when a client sends a request to it. In an ideal world, the request executes instantaneously, as shown in Figure 10.2.

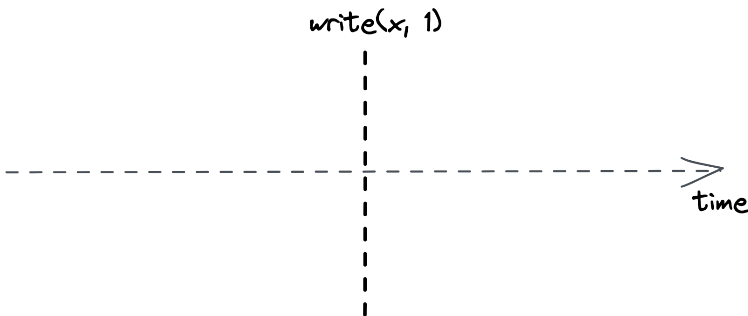

Figure 10.2: A write request executing instantaneously

But in reality, things are quite different — the request needs to reach the leader, which has to process it and send back a response to the client. As shown in Figure 10.3, these actions take time and are not instantaneous.

The best guarantee the system can provide is that the request executes somewhere between its invocation and completion time. You might think that this doesn't look like a big deal; after all, it's

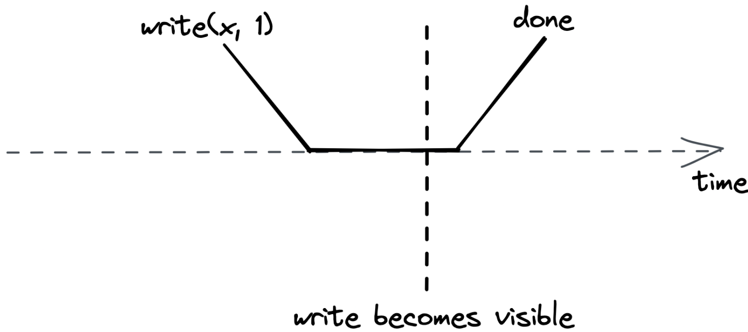

Figure 10.3: A write request can't execute instantaneously because it takes time to reach the leader and be executed.

what you are used to when writing single-threaded applications. For example, if you assign 42 to x and read its value immediately afterward, you expect to find 42 in there, assuming there is no other thread writing to the same variable. But when you deal with replicated systems, all bets are off. Let's see why that's the case.

In section 10.1, we looked at how Raft replicates the leader's state to its followers. Since only the leader can make changes to the state, any operation that modifies it needs to necessarily go through the leader. But what about reads? They don't necessarily have to go through the leader as they don't affect the system's state. Reads can be served by the leader, a follower, or a combination of leader and followers. If all reads have to go through the leader, the read throughput would be limited to that of a single process. But, if any follower can serve reads instead, then two clients, or observers, can have a different view of the system's state since followers can lag behind the leader.

Intuitively, there is a tradeoff between how consistent the observers' views of the system are and the system's performance and availability. To understand this relationship, we need to define precisely what we mean by consistency. We will do so with

the help of *consistency models*[11], which formally define the possible views the observers can have of the system's state.

10.3.1 Strong consistency

If clients send writes and reads exclusively to the leader, then every request appears to take place atomically at a very specific point in time as if there were a single copy of the data. No matter how many replicas there are or how far behind they are lagging, as long as the clients always query the leader directly, there is a single copy of the data from their point of view.

Because a request is not served instantaneously, and there is a single process that can serve it, the request executes somewhere between its invocation and completion time. By the time it completes, its side-effects are visible to all observers, as shown in Figure 10.4.

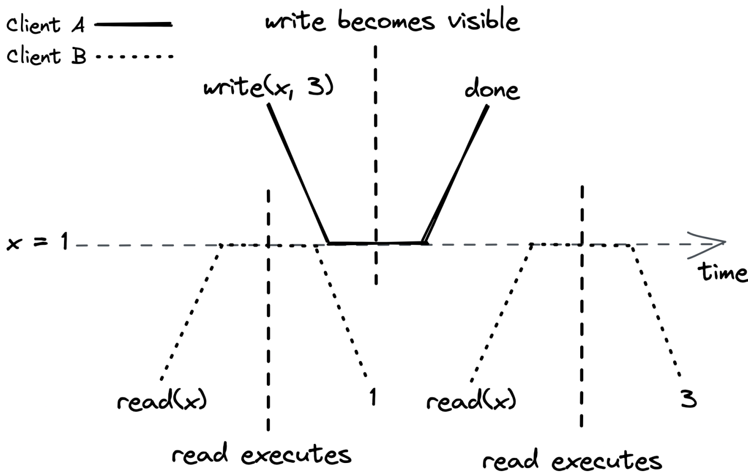

Figure 10.4: The side-effects of a strongly consistent operation are visible to all observers once it completes.

Since a request becomes visible to all other participants between its invocation and completion time, a real-time guarantee must

[11] "Consistency Models," https://jepsen.io/consistency

be enforced; this guarantee is formalized by a consistency model called *linearizability*[12], or *strong consistency*. Linearizability is the strongest consistency guarantee a system can provide for single-object requests.[13]

Unfortunately, the leader can't serve reads directly from its local state because by the time it receives a request from a client, it might no longer be the leader; so, if it were to serve the request, the system wouldn't be strongly consistent. The presumed leader first needs to contact a majority of replicas to confirm whether it still is the leader. Only then is it allowed to execute the request and send back a response to the client. Otherwise, it transitions to the follower state and fails the request. This confirmation step considerably increases the time required to serve a read.

10.3.2 Sequential consistency

So far, we have discussed serializing all reads through the leader. But doing so creates a single chokepoint, limiting the system's throughput. On top of that, the leader needs to contact a majority of followers to handle a read, which increases the time it takes to process a request. To increase the read performance, we could also allow the followers to handle requests.

Even though a follower can lag behind the leader, it will always receive new updates in the same order as the leader. For example, suppose one client only ever queries follower 1, and another only ever queries follower 2. In that case, the two clients will see the state evolving at different times, as followers are not perfectly in sync (see Figure 10.5).

The consistency model that ensures operations occur in the same order for all observers, but doesn't provide any real-time guarantee about when an operation's side-effect becomes visible to them, is called *sequential consistency*[14]. The lack of real-time guarantees is

[12] "Linearizability," https://jepsen.io/consistency/models/linearizable

[13] For example, this is the guarantee you would expect from a coordination service that manages leases.

[14] "Sequential Consistency," https://jepsen.io/consistency/models/sequential

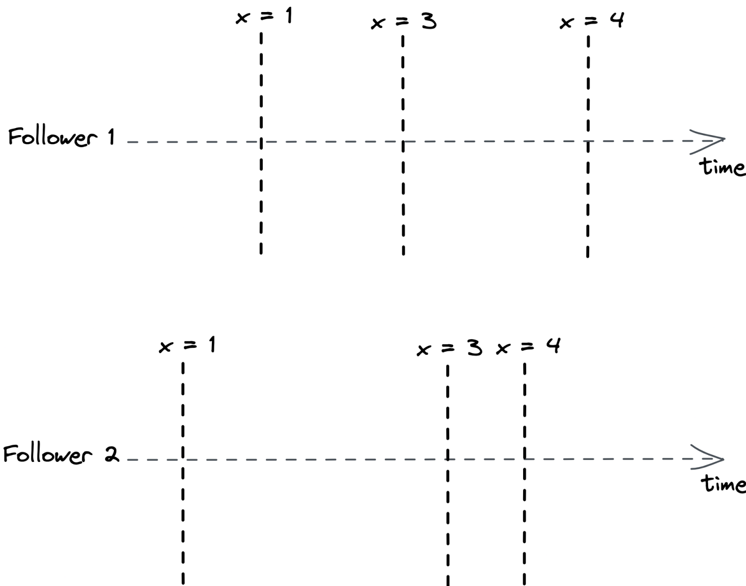

Figure 10.5: Although followers have a different view of the system's state, they process updates in the same order.

what differentiates sequential consistency from linearizability.

A producer/consumer system synchronized with a queue is an example of this model; a producer writes items to the queue, which a consumer reads. The producer and the consumer see the items in the same order, but the consumer lags behind the producer.

10.3.3 Eventual consistency

Although we managed to increase the read throughput, we had to pin clients to followers — if a follower becomes unavailable, the client loses access to the store. We could increase the availability by allowing the client to query any follower. But this comes at a steep price in terms of consistency. For example, say there are two followers, 1 and 2, where follower 2 lags behind follower 1. If a client queries follower 1 and then follower 2, it will see an earlier state, which can be very confusing. The only guarantee the client

has is that eventually all followers will converge to the final state if writes to the system stop. This consistency model is called *eventual consistency*.

It's challenging to build applications on top of an eventually consistent data store because the behavior is different from what we are used to when writing single-threaded applications. As a result, subtle bugs can creep up that are hard to debug and reproduce. Yet, in eventual consistency's defense, not all applications require linearizability. For example, an eventually consistent store is perfectly fine if we want to keep track of the number of users visiting a website, since it doesn't really matter if a read returns a number that is slightly out of date.

10.3.4 The CAP theorem

When a network partition happens, parts of the system become disconnected from each other. For example, some clients might no longer be able to reach the leader. The system has two choices when this happens; it can either:

- remain available by allowing clients to query followers that are reachable, sacrificing strong consistency;
- or guarantee strong consistency by failing reads that can't reach the leader.

This concept is expressed by the *CAP theorem*[15], which can be summarized as: "strong consistency, availability and partition tolerance: pick two out of three." In reality, the choice really is only between strong consistency and availability, as network faults are a given and can't be avoided.

Confusingly enough, the CAP theorem's definition of availability requires that every request *eventually* receives a response. But in real systems, achieving perfect availability is impossible. Moreover, a very slow response is just as bad as one that never occurs. So, in other words, many highly-available systems can't be con-

[15] "Perspectives on the CAP Theorem," https://groups.csail.mit.edu/tds/papers/Gilbert/Brewer2.pdf

sidered available as defined by the CAP theorem. Similarly, the theorem's definition of consistency and partition tolerance is very precise, limiting its practical applications.[16] A more useful way to think about the relationship between availability and consistency is as a spectrum. And so, for example, a strongly consistent and partition-tolerant system as defined by the CAP theorem occupies just one point in that spectrum.[17]

Also, even though network partitions can happen, they are usually rare within a data center. But, even in the absence of a network partition, there is a tradeoff between consistency and *latency* (or performance). The stronger the consistency guarantee is, the higher the latency of individual operations must be. This relationship is expressed by the *PACELC theorem*[18], an extension to the CAP theorem. It states that in case of network partitioning (P), one has to choose between availability (A) and consistency (C), but else (E), even when the system is running normally in the absence of partitions, one has to choose between latency (L) and consistency (C). In practice, the choice between latency and consistency is not binary but rather a spectrum.

This is why some off-the-shelf distributed data stores come with counter-intuitive consistency guarantees in order to provide high availability and performance. Others have knobs that allow you to choose whether you want better performance or stronger consistency guarantees, like Azure's Cosmos DB[19] and Cassandra[20].

Another way to interpret the PACELC theorem is that there is a tradeoff between the amount of coordination required and performance. One way to design around this fundamental limitation is

[16] "A Critique of the CAP Theorem," https://www.cl.cam.ac.uk/research/dtg/www/files/publications/public/mk428/cap-critique.pdf

[17] "CAP Theorem: You don't need CP, you don't want AP, and you can't have CA," https://www.youtube.com/watch?v=hUd_9FENShA

[18] "Consistency Tradeoffs in Modern Distributed Database System Design," https://en.wikipedia.org/wiki/PACELC_theorem

[19] "Consistency levels in Azure Cosmos DB," https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels

[20] "Apache Cassandra: How is the consistency level configured?," https://docs.datastax.com/en/cassandra-oss/3.0/cassandra/dml/dmlConfigConsistency.html

to move coordination away from the critical path. For example, earlier we discussed that for a read to be strongly consistent, the leader has to contact a majority of followers. That coordination tax is paid for each read! In the next section, we will explore a different replication protocol that moves this cost away from the critical path.

10.4 Chain replication

Chain replication[21] is a widely used replication protocol that uses a very different topology from leader-based replication protocols like Raft. In chain replication, processes are arranged in a chain. The leftmost process is referred to as the chain's *head*, while the rightmost one is the chain's *tail*.

Clients send writes exclusively to the head, which updates its local state and forwards the update to the next process in the chain. Similarly, that process updates its state and forwards the change to its successor until it eventually reaches the tail.

When the tail receives an update, it applies it locally and sends an acknowledgment to its predecessor to signal that the change has been committed. The acknowledgment flows back to the head, which can then reply to the client that the write succeeded[22].

Client reads are served exclusively by the tail, as shown in Fig 10.6. In the absence of failures, the protocol is strongly consistent as all writes and reads are processed one at a time by the tail. But what happens if a process in the chain fails?

Fault tolerance is delegated to a dedicated component, the configuration manager or *control plane*. At a high level, the control plane monitors the chain's health, and when it detects a faulty process, it removes it from the chain. The control plane ensures that there is a

[21] "Chain Replication for Supporting High Throughput and Availability," https://www.cs.cornell.edu/home/rvr/papers/OSDI04.pdf

[22] This is slightly different from the original chain replication paper since it's based on CRAQ, an extension of the original protocol; see "Object Storage on CRAQ," https://www.usenix.org/legacy/event/usenix09/tech/full_papers/terrace/terrace.pdf.

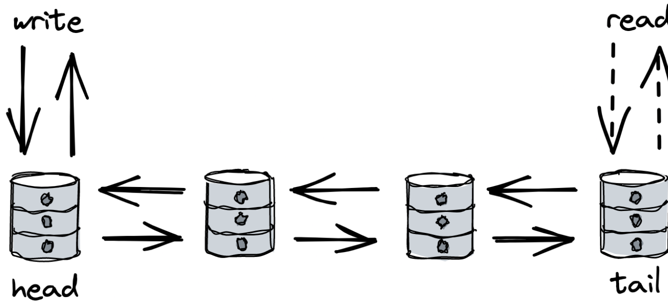

Figure 10.6: Writes propagate through all processes in the chain, while reads are served exclusively by the tail.

single view of the chain's topology that every process agrees with. For this to work, the control plane needs to be fault-tolerant, which requires state machine replication (e.g., Raft). So while the chain can tolerate up to $N - 1$ processes failing, where N is the chain's length, the control plane can only tolerate $\frac{C}{2}$ failures, where C is the number of replicas that make up the control plane.

There are three failure modes in chain replication: the head can fail, the tail can fail, or an intermediate process can fail. If the head fails, the control plane removes it by reconfiguring its successor to be the new head and notifying clients of the change. If the head committed a write to its local state but crashed before forwarding it downstream, no harm is done. Since the write didn't reach the tail, the client that issued it hasn't received an acknowledgment for it yet. From the client's perspective, it's just a request that timed out and needs to be retried. Similarly, no other client will have seen the write's side effects since it never reached the tail.

If the tail fails, the control plane removes it and makes its predecessor the chain's new tail. Because all updates that the tail has received must necessarily have been received by the predecessor as well, everything works as expected.

If an intermediate process X fails, the control plane has to link X's predecessor with X's successor. This case is a bit trickier to handle

since X might have applied some updates locally but failed before forwarding them to its successor. Therefore, X's successor needs to communicate to the control plane the sequence number of the last committed update it has seen, which is then passed to X's predecessor to send the missing updates downstream.

Chain replication can tolerate up to $N - 1$ failures. So, as more processes in the chain fail, it can tolerate fewer failures. This is why it's important to replace a failing process with a new one. This can be accomplished by making the new process the tail of the chain after syncing it with its predecessor.

The beauty of chain replication is that there are only a handful of simple failure modes to consider. That's because for a write to commit, it needs to reach the tail, and consequently, it must have been processed by every process in the chain. This is very different from a quorum-based replication protocol like Raft, where only a subset of replicas may have seen a committed write.

Chain replication is simpler to understand and more performant than leader-based replication since the leader's job of serving client requests is split among the head and the tail. The head sequences writes by updating its local state and forwarding updates to its successor. Reads, however, are served by the tail, and are interleaved with updates received from its predecessor. Unlike in Raft, a read request from a client can be served immediately from the tail's local state without contacting the other replicas first, which allows for higher throughputs and lower response times.

However, there is a price to pay in terms of write latency. Since an update needs to go through all the processes in the chain before it can be considered committed, a single slow replica can slow down all writes. In contrast, in Raft, the leader only has to wait for a majority of processes to reply and therefore is more resilient to transient degradations. Additionally, if a process isn't available, chain replication can't commit writes until the control plane detects the problem and takes the failing process out of the chain. In Raft instead, a single process failing doesn't stop writes from being committed since only a quorum of processes is needed to make

progress.

That said, chain replication allows write requests to be pipelined, which can significantly improve throughput. Moreover, read throughput can be further increased by distributing reads across replicas while still guaranteeing linearizability. The idea is for replicas to store multiple versions of an object, each including a version number and a dirty flag. Replicas mark an update as dirty as it propagates from the head to the tail. Once the tail receives it, it's considered committed, and the tail sends an acknowledgment back along the chain. When a replica receives an acknowledgment, it marks the corresponding version as clean. Now, when a replica receives a read request for an object, it will immediately serve it if the latest version is clean. If not, it first contacts the tail to request the latest committed version (see Fig 10.7).

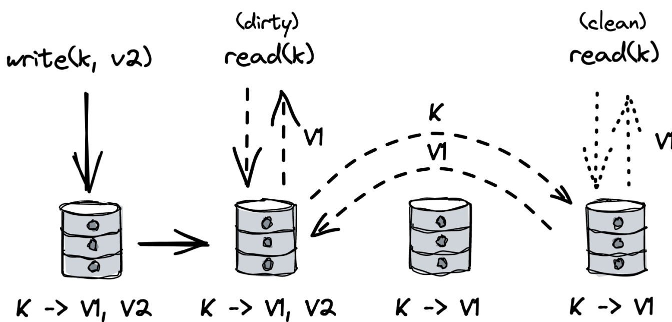

Figure 10.7: A dirty read can be served by any replica with an additional request to the tail to guarantee strong consistency.

As discussed in chapter 9, a leader introduces a scalability bottleneck. But in chain replication, the data plane (i.e., the part of the system that handles individual client requests on the critical path) doesn't need a leader to do its job since it's not concerned with failures — its sole focus is throughput and efficiency. On the contrary, the control plane needs a leader to implement state machine replication, but that's required exclusively to handle the occasional failure and doesn't affect client requests on the critical path. An-

other way to think about this is that chain replication reduces the amount of coordination needed for each client request. In turn, this increases the data plane's capacity to handle load. For this reason, splitting the data plane from the control plane (i.e., the configuration management part) is a common pattern in distributed systems. We will talk in more detail about this in chapter 22.

You might be wondering at this point whether it's possible to replicate data without needing consensus[23] at all to improve performance further. In the next chapter, we will try to do just that.

[23] required for state machine replication

Chapter 11

Coordination avoidance

Another way of looking at state machine replication is as a system that requires two main ingredients:

- a *broadcast protocol* that guarantees every replica receives the same updates in the same order even in the presence of faults (aka *fault-tolerant total order broadcast*),
- and a deterministic function that handles updates on each replica.

Unsurprisingly, implementing a fault-tolerant total order broadcast protocol is what makes state machine replication hard to solve since it requires consensus[1]. More importantly, the need for a total order creates a scalability bottleneck since updates need to be processed sequentially by a single process (e.g., the leader in Raft). Also, total order broadcast isn't available during network partitions as the CAP theorem applies[2] to it as well[3].

In this chapter, we will explore a form of replication that doesn't

[1]Total order broadcast is equivalent to consensus, see "Unreliable Failure Detectors for Reliable Distributed Systems," https://www.cs.utexas.edu/~lorenzo/corsi/cs380d/papers/p225-chandra.pdf

[2]"Perspectives on the CAP Theorem," https://groups.csail.mit.edu/tds/papers/Gilbert/Brewer2.pdf

[3]Consensus is harder to solve than implementing a linearizable read/write register, which is what the CAP theorem uses to define consistency.

require a total order but still comes with useful guarantees. But first, we need to talk about broadcast protocols.

11.1 Broadcast protocols

Network communication over wide area networks, like the internet, only offers point-to-point (unicast) communication protocols, like TCP. But to deliver a message to a group of processes, a broadcast protocol is needed (multicast). This means we have to somehow build a multicast protocol on top of a unicast one. The challenge here is that multicast needs to support multiple senders and receivers that can crash at any time.

A broadcast protocol is characterized by the guarantees it provides. *Best-effort broadcast* guarantees that if the sender doesn't crash, the message is delivered to all non-faulty processes in a group. A simple way to implement it is to send the message to all processes in a group one by one over reliable links (see Fig 11.1). However, if, for example, the sender fails mid-way, some processes will never receive the message.

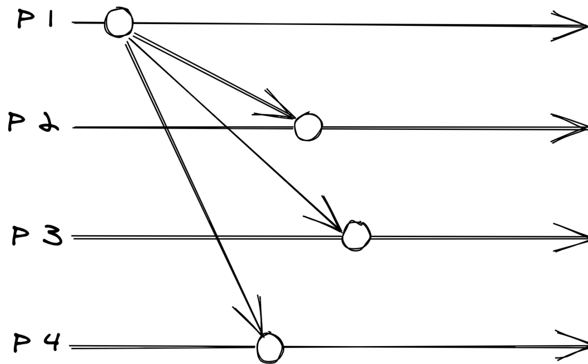

Figure 11.1: Best-effort broadcast

Unlike best-effort broadcast, *reliable broadcast* guarantees that the message is eventually delivered to all non-faulty processes in the group, even if the sender crashes before the message has been fully

delivered. One way to implement reliable broadcast is to have each process retransmit the message to the rest of the group the first time it is delivered (see Fig 11.2). This approach is also known as *eager reliable broadcast*. Although it guarantees that all non-faulty processes eventually receive the message, it's costly as it requires sending the message N^2 times for a group of N processes.

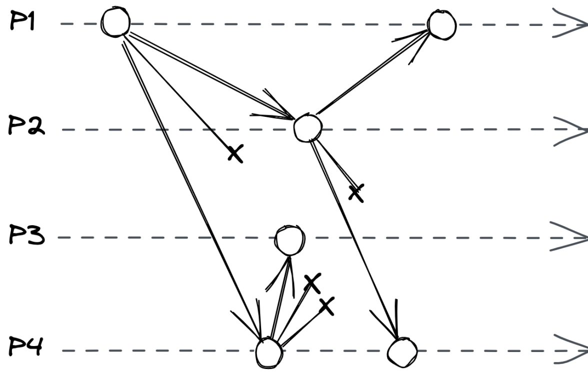

Figure 11.2: Eager reliable broadcast

The number of messages can be reduced by retransmitting a message only to a random subset of processes (e.g., 2 as in Fig 11.3). This implementation is referred to as a *gossip broadcast protocol*[4] as it resembles how rumors spread. Because it's a probabilistic protocol, it doesn't guarantee that a message will be delivered to all processes. That said, it's possible to make that probability negligible by tuning the protocol's parameters. Gossip protocols are particularly useful when broadcasting to a large number of processes where a deterministic protocol just wouldn't scale.

Although reliable broadcast protocols guarantee that messages are delivered to all non-faulty processes in a group, they don't make any guarantees about their order. For example, two processes could receive the same messages but in a different order. *Total order broadcast* is a reliable broadcast abstraction that builds upon the guarantees offered by reliable broadcast and additionally

[4] "Gossip protocol," https://en.wikipedia.org/wiki/Gossip_protocol

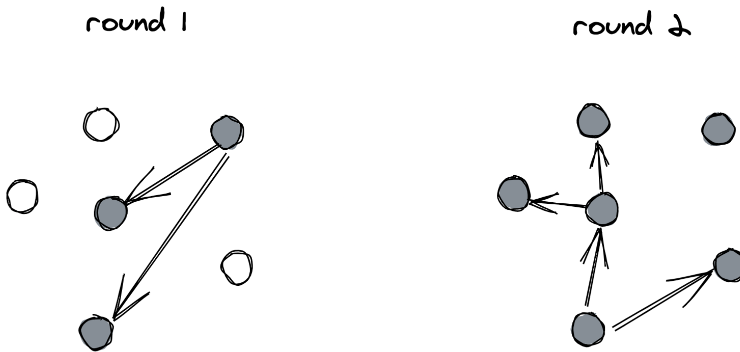

Figure 11.3: Gossip broadcast

ensures that messages are delivered in the same order to all processes. As discussed earlier, a fault-tolerant implementation requires consensus.

11.2 Conflict-free replicated data types

Now, here's an idea: if we were to implement replication with a broadcast protocol that doesn't guarantee total order, we wouldn't need to serialize writes through a single leader, but instead could allow any replica to accept writes. But since replicas might receive messages in different orders, they will inevitably diverge. So, for the replication to be useful, the divergence can only be temporary, and replicas eventually have to converge to the same state. This is the essence of eventual consistency.

More formally, eventual consistency requires:

- *eventual delivery* — the guarantee that every update applied at a replica is eventually applied at all replicas,
- and *convergence* — the guarantee that replicas that have applied the same updates *eventually* reach the same state.

Using a broadcast protocol that doesn't deliver messages in the same order across all replicas will inevitably lead to divergence

(see Fig 11.4). One way to reconcile conflicting writes is to use consensus to make a decision that all replicas need to agree with.

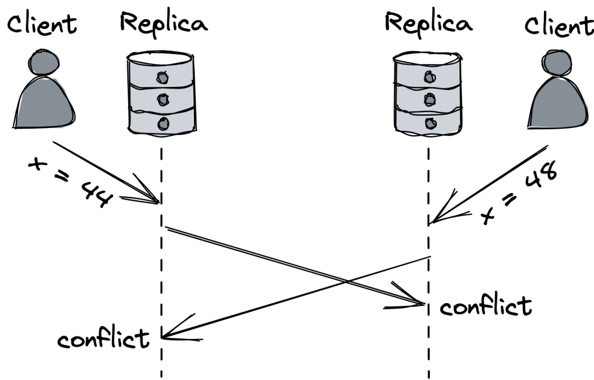

Figure 11.4: The same object is updated simultaneously by different clients at different replicas, leading to conflicts.

This solution has better availability and performance than the one using total order broadcast, since consensus is only required to reconcile conflicts and can happen off the critical path. But getting the reconciliation logic right isn't trivial. So is there a way for replicas to solve conflicts without using consensus at all?

Well, if we can define a deterministic outcome for any potential conflict (e.g., the write with the greatest timestamp always wins), there wouldn't be any conflicts, by design. Therefore consensus wouldn't be needed to reconcile replicas. Such a replication strategy offers stronger guarantees than plain eventual consistency, i.e.:

- *eventual delivery* — the same guarantee as in eventual consistency,
- and *strong convergence* — the guarantee that replicas that have executed the same updates *have* the same state (i.e., every update is immediately persisted).

This variation of eventual consistency is also called *strong even-*

tual consistency[5]. With it, we can build systems that are available, (strongly eventual) consistent, and also partition tolerant.

Which conditions are required to guarantee that replicas strongly converge? For example, suppose we replicate an object across N replicas, where the object is an instance of some data type that supports *query* and *update* operations (e.g., integer, string, set, etc.).

A client can send an update or query operation to any replica, and:

- when a replica receives a query, it immediately replies using the local copy of the object;
- when a replica receives an update, it first applies it to the local copy of the object and then broadcasts the updated object to all replicas;
- and when a replica receives a broadcast message, it *merges* the object in the message with its own.

It can be shown that each replica will converge to the same state if:

- the object's possible states form a semilattice, i.e., a set that contains elements that can be partially ordered;
- and the merge operation returns the least upper bound between two objects' states (and therefore is idempotent, commutative, and associative).

A data type that has these properties is also called a convergent replicated data type[6], which is part of the family of *conflict-free replicated data types* (CRDTs). This sounds a lot more complicated than it actually is.

For example, suppose we are working with integer objects (which can be partially ordered), and the merge operation takes the maximum of two objects (least upper bound). It's easy to see how replicas converge to the global maximum in this case, even if requests are delivered out of order and/or multiple times across replicas.

---

[5] "Strong Eventual Consistency and Conflict-free Replicated Data Types," https://www.microsoft.com/en-us/research/video/strong-eventual-consistency-and-conflict-free-replicated-data-types/

[6] "Conflict-free Replicated Data Types," https://hal.inria.fr/inria-00609399v1/document

Although we have assumed the use of a reliable broadcast protocol so far, replicas could even use an unreliable protocol to implement broadcast as long as they periodically exchange and merge their states to ensure that they eventually converge (aka an *anti-entropy mechanism*, we will see some examples in section 11.3). Of course, periodic state exchanges can be expensive if done naively.

There are many data types that are designed to converge when replicated, like registers, counters, sets, dictionaries, and graphs. For example, a register is a memory cell storing some opaque sequence of bytes that supports an assignment operation to overwrite its state. To make a register convergent, we need to define a partial order over its values and a merge operation. There are two common register implementations that meet these requirements: last-writer-wins (LWW) and multi-value (MV).

A *last-writer-wins* register associates a timestamp with every update to make updates totally orderable. The timestamp could be composed of a Lamport timestamp to preserve the *happened-before* relationship among updates and a replica identifier to ensure there are no ties. When a replica receives an update request from a client, it generates a new timestamp and updates the register's state with that and the new value; finally, it broadcasts the state and timestamp to all replicas. When a replica receives a register state from a peer, it merges it with its local copy by taking the one with the greater timestamp and discarding the other (see Fig 11.5).

The main issue with LWW registers is that conflicting updates that happen concurrently are handled by taking the one with the greater timestamp, which might not always make sense. An alternative way of handling conflicts is to keep track of all concurrent updates and return them to the client application, which can handle conflicts however it sees fit. This is the approach taken by the *multi-value* register. To detect concurrent updates, replicas tag each update with a vector clock timestamp[7] and the merge

[7] In practice, *version vectors* are used to compare the state of different replicas that only keep track of events that change the state of replicas, see "Detection of Mutual Inconsistency in Distributed Systems," https://pages.cs.wisc.edu/~remzi/Classes/739/Fall2017/Papers/parker83detection.pdf.

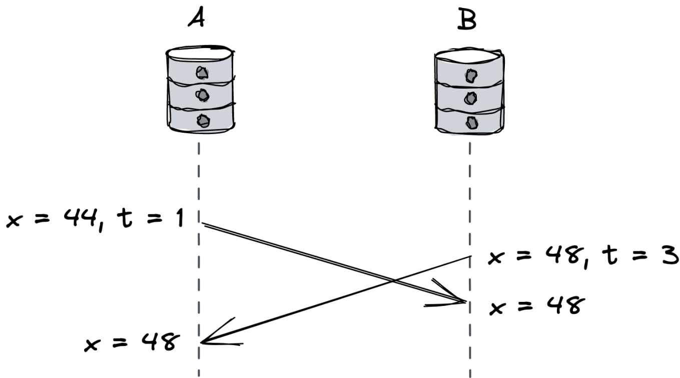

Figure 11.5: Last-writer wins register

operation returns the union of all concurrent updates (see Fig 11.6).

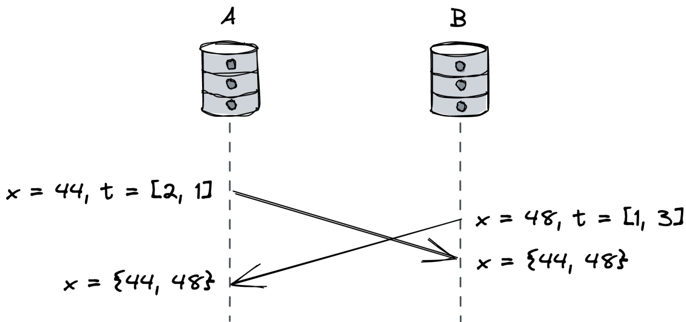

Figure 11.6: Multi-value register

The beauty of CRDTs is that they compose. So, for example, you can build a convergent key-value store by using a dictionary of LWW or MV registers. This is the approach followed by *Dynamo-style* data stores.

11.3 Dynamo-style data stores

Dynamo[8] is arguably the best-known design of an eventually consistent and highly available key-value store. Many other data stores have been inspired by it, like Cassandra[9] and Riak KV[10].

In Dynamo-style data stores, every replica can accept write and read requests. When a client wants to write an entry to the data store, it sends the request to all N replicas in parallel but waits for an acknowledgment from just W replicas (a write quorum). Similarly, when a client wants to read an entry from the data store, it sends the request to all replicas but waits just for R replies (a read quorum) and returns the most recent entry to the client. To resolve conflicts, entries behave like LWW or MV registers depending on the implementation flavor.

When $W + R > N$, the write quorum and the read quorum must intersect with each other, so at least one read will return the latest version (see Fig 11.7). This doesn't guarantee linearizability on its own, though. For example, if a write succeeds on less than W replicas and fails on the others, replicas are left in an inconsistent state, and some clients might read the latest version while others don't. To avoid this inconsistency, the writes need to be bundled into an atomic transaction. We will talk more about transactions in chapter 12.

Typically W and R are configured to be majority quorums, i.e., quorums that contain more than half the number of replicas. That said, other combinations are possible, and the data store's read and write throughput depend on how large or small R and W are. For example, a read-heavy workload benefits from a smaller R; however, this makes writes slower and less available (assuming $W + R > N$). Alternatively, both W and R can be configured to be very small (e.g., $W = R = 1$) for maximum performance at the expense of consistency ($W + R < N$).

[8] "Dynamo: Amazon's Highly Available Key-value Store," https://www.allthingsdistributed.com/files/amazon-dynamo-sosp2007.pdf

[9] "Cassandra: Open Source NoSQL Database," https://cassandra.apache.org/

[10] "Riak KV: A distributed NoSQL key-value database," https://riak.com/products/riak-kv/

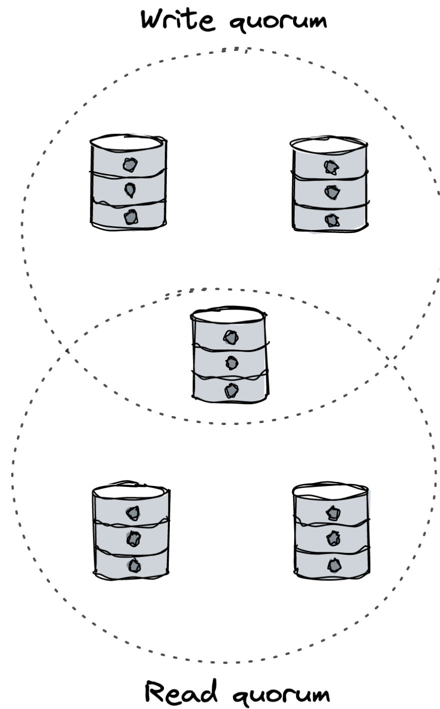

Figure 11.7: Intersecting write and read quorums

One problem with this approach is that a write request sent to a replica might never make it to the destination. In this case, the replica won't converge, no matter how long it waits. To ensure that replicas converge, two anti-entropy mechanisms are used: read-repair and replica synchronization. So another way to think about quorum replication is as a best-effort broadcast combined with anti-entropy mechanisms to ensure that all changes propagate to all replicas.

Read repair is a mechanism that clients implement to help bring replicas back in sync whenever they perform a read. As mentioned earlier, when a client executes a read, it waits for R replies. Now, suppose some of these replies contain older entries. In that case, the client can issue a write request with the latest entry to the out-

of-sync replicas. Although this approach works well for frequently read entries, it's not enough to guarantee that all replicas will eventually converge.

Replica synchronization is a continuous background mechanism that runs on every replica and periodically communicates with others to identify and repair inconsistencies. For example, suppose replica X finds out that it has an older version of key K than replica Y. In that case, it will retrieve the latest version of K from Y. To detect inconsistencies and minimize the amount of data exchanged, replicas can exchange Merkle tree hashes[11] with a gossip protocol.

11.4 The CALM theorem

At this point, you might be wondering how you can tell whether an application requires coordination, such as consensus, and when it doesn't. The CALM theorem[12] states that a program has a consistent, coordination-free distributed implementation if and only if it is *monotonic*.

Intuitively, a program is monotonic if new inputs further refine the output and can't take back any prior output. A program that computes the union of a set is a good example of that — once an element (input) is added to the set (output), it can't be removed. Similarly, it can be shown that CRDTs are monotonic.

In contrast, in a non-monotonic program, a new input can retract a prior output. For example, variable assignment is a non-monotonic operation since it overwrites the variable's prior value.

A monotonic program can be consistent, available, and partition tolerant all at once. However, consistency in CALM doesn't refer to linearizability, the *C* in CAP. Linearizability is narrowly focused on the consistency of reads and writes. Instead, CALM focuses on

[11] "Merkle tree," https://en.wikipedia.org/wiki/Merkle_tree

[12] "Keeping CALM: When Distributed Consistency is Easy," https://arxiv.org/pdf/1901.01930.pdf

the consistency of the program's output[13]. In CALM, a consistent program is one that produces the same output no matter in which order the inputs are processed and despite any conflicts; it doesn't say anything about the consistency of reads and writes.

For example, say you want to implement a counter. If all you have at your disposal are write and read operations, then the order of the operations matters:

```
write(1), write(2), write(3) => 3
```

but:

```
write(3), write(1), write(2) => 2
```

In contrast, if the program has an abstraction for counters that supports an increment operation, you can reorder the operations any way you like without affecting the result:

```
increment(1), increment(1), increment(1) => 3
```

In other words, consistency based on reads and writes can limit the solution space[14], since it's possible to build systems that are consistent at the application level, but not in terms of reads and writes at the storage level.

CALM also identifies programs that can't be consistent because they are not monotonic. For example, a vanilla register/variable assignment operation is not monotonic as it invalidates whatever value was stored there before. But, by combining the assignment operation with a logical clock, it's possible to build a monotonic implementation, as we saw earlier when discussing LWW and MV registers.

11.5 Causal consistency

So we understand now how eventual consistency can be used to implement monotonic applications that are consistent, available,

[13] Consistency can have different meanings depending on the context; make sure you know precisely what it refers to when you encounter it.

[14] "Building on Quicksand," https://dsf.berkeley.edu/cs286/papers/quicksand-cidr2009.pdf

and partition-tolerant. Unfortunately, there are many applications for which its guarantees are not sufficient. For example, eventual consistency doesn't guarantee that an operation that *happened-before* another is observed in the correct order by replicas. Suppose you upload a picture to a social network and then add it to a gallery. With eventual consistency, the gallery may reference the image before it becomes available, causing a missing image placeholder to appear in its stead.

One of the main benefits of strong consistency is that it preserves the *happened-before* order among operations, which guarantees that the cause happens before the effect. So, in the previous example, the reference to the newly added picture in the gallery is guaranteed to become visible only after the picture becomes available.

Surprisingly, to preserve the *happened-before* order (causal order) among operations, we don't need to reach for strong consistency, since we can use a weaker consistency model called *causal consistency*[15]. This model is weaker than strong consistency but stronger than eventual consistency, and it's particularly attractive for two reasons:

- For many applications, causal consistency is "consistent enough" and easier to work with than eventual consistency.
- Causal consistency is provably[16] the strongest consistency model that enables building systems that are also available and partition tolerant.

Causal consistency imposes a partial order on the operations. The simplest definition requires that processes agree on the order of causally related operations but can *disagree* on the order of unrelated ones. You can take any two operations, and either one *happened-before* the other, or they are concurrent and therefore can't be ordered. This is the main difference from strong consistency, which imposes a *global* order that all processes agree with.

For example, suppose a process updates an entry in a key-value

[15] "Causal Consistency," https://jepsen.io/consistency/models/causal
[16] "Consistency, Availability, and Convergence," https://apps.cs.utexas.edu/tech_reports/reports/tr/TR-2036.pdf

store (operation A), which is later read by another process (operation B) that consequently updates another entry (operation C). In that case, all processes in the system have to agree that A *happened-before* C. In contrast, if two operations, X and Y, happen concurrently and neither *happened-before* the other, some processes may observe X before Y and others Y before X.

Let's see how we can use causal consistency to build a replicated data store that is available under network partitions. We will base our discussion on "Clusters of Order-Preserving Servers" (COPS[17]), a key-value store that delivers causal consistency across geographically distributed clusters. In COPS, a cluster is set up as a strongly consistent partitioned data store, but for simplicity, we will treat it as a single logical node without partitions.[18] Later, in chapter 16, we will discuss partitioning at length.

First, let's define a variant of causal consistency called *causal+* in which there is no disagreement (conflict) about the order of unrelated operations. Disagreements are problematic since they cause replicas to diverge forever. To avoid them, LWW registers can be used as values to ensure that all replicas converge to the same state in the presence of concurrent writes. An LWW register is composed of an object and a logical timestamp, which represents its version.

In COPS, any replica can accept read and write requests, and clients send requests to their closest replica (local replica). When a client sends a read request for a key to its local replica, the latter replies with the most recent value available locally. When the client receives the response, it adds the version (logical timestamp) of the value it received to a local key-version dictionary used to keep track of *dependencies*.

When a client sends a write to its local replica, it adds a copy of

[17] "Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS," https://www.cs.princeton.edu/~mfreed/docs/cops-sosp11.pdf

[18] COPS can track causal relationships between partitions (and therefore nodes), unlike simpler approaches using version vectors, which limit causality tracking to the set of keys that a single node can store (see "Session Guarantees for Weakly Consistent Replicated Data," https://www.cs.utexas.edu/users/dahlin/Classes/GradOS/papers/SessionGuaranteesPDIS.pdf).

the dependency dictionary to the request. The replica assigns a version to the write, applies the change locally, and sends an acknowledgment back to the client with the version assigned to it. It can apply the change locally, even if other clients updated the key in the meantime, because values are represented with LWW registers. Finally, the update is broadcast asynchronously to the other replicas.

When a replica receives a replication message for a write, it doesn't apply it locally immediately. Instead, it first checks whether the write's dependencies have been committed locally. If not, it waits until the required versions appear. Finally, once all dependencies have been committed, the replication message is applied locally. This behavior guarantees causal consistency (see Fig 11.8).

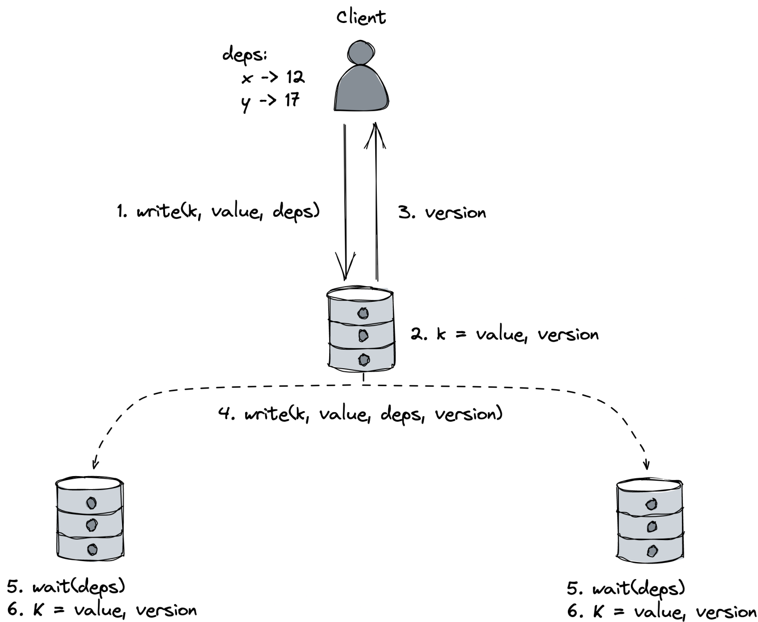

Figure 11.8: A causally consistent implementation of a key-value store

If a replica fails, the data store continues to be available as any replica can accept writes. There is a possibility that a replica could

fail after committing an update locally but before broadcasting it, resulting in data loss. In COPS' case, this tradeoff is considered acceptable to avoid paying the price of waiting for one or more long-distance requests to remote replicas before acknowledging a client write.

11.6 Practical considerations

To summarize, in the previous chapters we explored different ways to implement replication and learned that there is a tradeoff between consistency and availability/performance. In other words, to build scalable and available systems, coordination needs to be minimized.

This tradeoff is present in any large-scale system, and some even have knobs that allow you to control it. For example, Azure Cosmos DB is a fully managed scalable NoSQL database that enables developers to choose among 5 different consistency models, ranging from eventual consistency to strong consistency[19], where weaker consistency models have higher throughputs than stronger ones.

[19]"Azure Cosmos DB: Pushing the frontier of globally distributed databases," https://azure.microsoft.com/en-gb/blog/azure-cosmos-db-pushing-the-frontier-of-globally-distributed-databases/

Chapter 12

Transactions

Transactions[1] provide the illusion that either all the operations within a group complete successfully or none of them do, as if the group were a single atomic operation. If you have used a relational database such as MySQL or PostgreSQL in the past, you should be familiar with the concept.

If your application exclusively updates data within a single relational database, then bundling some changes into a transaction is straightforward. On the other hand, if your system needs to atomically update data that resides in multiple data stores, the operations need to be wrapped into a *distributed* transaction, which is a lot more challenging to implement. This is a fairly common scenario in microservice architectures where a service needs to interact with other services to handle a request, and each service has its own separate data store. In this chapter, we explore various solutions to this problem.

---

[1] "Transaction processing," https://en.wikipedia.org/wiki/Transaction_processing

12.1 ACID

Consider a money transfer from one bank account to another. If the withdrawal succeeds, but the deposit fails for some reason, the funds need to be deposited back into the source account. In other words, the transfer needs to execute atomically; either both the withdrawal and the deposit succeed, or in case of a failure, neither do. To achieve that, the withdrawal and deposit need to be wrapped in an inseparable unit of change: a *transaction*.

In a traditional relational database, a transaction is a group of operations for which the database guarantees a set of properties, known as *ACID*:

- **A**tomicity guarantees that partial failures aren't possible; either all the operations in the transactions complete successfully, or none do. So if a transaction begins execution but fails for whatever reason, any changes it made must be undone. This needs to happen regardless of whether the transaction itself failed (e.g., divide by zero) or the database crashed mid way.
- **C**onsistency guarantees that the application-level invariants must always be true. In other words, a transaction can only transition a database from a correct state to another correct state. How this is achieved is the responsibility of the application developer who defines the transaction. For example, in a money transfer, the invariant is that the sum of money across the accounts is the same after the transfer, i.e., money can't be destroyed or created. Confusingly, the "C" in ACID has nothing to do with the consistency models we talked about so far, and according to Joe Hellerstein, it was tossed in to make the acronym work[2]. Therefore, we will safely ignore this property in the rest of the chapter.
- **I**solation guarantees that a transaction appears to run in isolation as if no other transactions are executing, i.e., the concurrent execution of transactions doesn't cause any race con-

[2] "When is"ACID" ACID? Rarely." http://www.bailis.org/blog/when-is-acid-acid-rarely/

ditions.

- **D**urability guarantees that once the database commits the transaction, the changes are persisted on durable storage so that the database doesn't lose the changes if it subsequently crashes. In the way I described it, it sounds like the job is done once the data is persisted to a storage device. But, we know better by now, and replication[3] is required to ensure durability in the presence of storage failures.

Transactions relieve developers from a whole range of possible failure scenarios so that they can focus on the actual application logic rather than handling failures. But to understand how distributed transactions work, we first need to discuss how centralized, non-distributed databases implement transactions.

12.2 Isolation

The easiest way to guarantee that no transaction interferes with another is to run them serially one after another (e.g., using a global lock). But, of course, that would be extremely inefficient, which is why in practice transactions run concurrently. However, a group of concurrently running transactions accessing the same data can run into all sorts of race conditions, like dirty writes, dirty reads, fuzzy reads, and phantom reads:

- A *dirty write* happens when a transaction overwrites the value written by another transaction that hasn't committed yet.
- A *dirty read* happens when a transaction observes a write from a transaction that hasn't completed yet.
- A *fuzzy read* happens when a transaction reads an object's value twice but sees a different value in each read because another transaction updated the value between the two reads.
- A *phantom read* happens when a transaction reads a group of objects matching a specific condition, while another transaction concurrently adds, updates, or deletes objects match-

[3] see Chapter 10

ing the same condition. For example, if one transaction is summing all employees' salaries while another deletes some employee records simultaneously, the final sum will be incorrect at commit time.

To protect against these race conditions, a transaction needs to be isolated from others. An *isolation level* protects against one or more types of race conditions and provides an abstraction that we can use to reason about concurrency. The stronger the isolation level is, the more protection it offers against race conditions, but the less performant it is.

An isolation level is defined based on the type of race conditions it forbids, as shown in Figure 12.1.

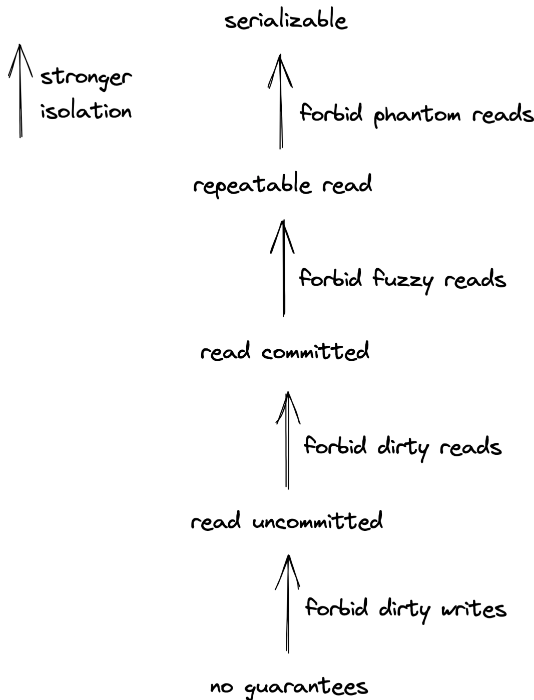

Figure 12.1: Isolation levels define which race conditions they forbid.

Serializability is the only isolation level that isolates against all pos-

sible race conditions. It guarantees that executing a group of transactions has the same side effects as if the transactions run serially (one after another) in *some* order. For example, suppose we have two concurrent transactions, X and Y, and transaction X commits before transaction Y. A serializable system guarantees that even though their operations are interleaved, they appear to run after the other, i.e., X before Y or Y before X (even if Y committed after!). To add a real-time requirement on the order of transactions, we need a stronger isolation level: *strict serializability*. This level combines serializability with the real-time guarantees of linearizability so that when a transaction completes, its side effects become immediately visible to all future transactions.

(Strict) serializability is slow as it requires coordination, which creates contention in the system. For example, a transaction may be forced to wait for other transactions. In some cases, it may even be forced to abort because it can no longer be executed as part of a serializable execution. Because not all applications require the guarantees that serializability provides, data stores allow developers to use weaker isolation levels. As a rule of thumb, we need to consciously decide which isolation level to use and understand its implications, or the data store will silently make that decision for us; for example, PostgreSQL's default isolation[4] is read committed. So, if in doubt, choose strict serializability.

There are more isolation levels and race conditions than the ones we discussed here. Jepsen[5] provides a good formal reference for the existing isolation levels, how they relate to one another, and which guarantees they offer. Although vendors typically document the isolation levels of their products, these specifications don't always match[6] the formal definitions.

Now that we know what serializability is, the challenge becomes maximizing concurrency while still preserving the appearance of serial execution. The concurrency strategy is defined by a *concur-*

[4] "PostgreSQL Transaction Isolation," https://www.postgresql.org/docs/12/transaction-iso.html
[5] "Consistency Models," https://jepsen.io/consistency
[6] "Jepsen Analyses," https://jepsen.io/analyses

rency control protocol, and there are two categories of protocols that guarantee serializability: pessimistic and optimistic.

12.2.1 Concurrency control

A *pessimistic* protocol uses locks to block other transactions from accessing an object. The most commonly used protocol is *two-phase locking*[7] (2PL). 2PL has two types of locks, one for reads and one for writes. A read lock can be shared by multiple transactions that access the object in read-only mode, but it blocks transactions trying to acquire a write lock. A write lock can be held only by a single transaction and blocks anyone trying to acquire either a read or write lock on the object. The locks are held by a *lock manager* that keeps track of the locks granted so far, the transactions that acquired them, and the transactions waiting for them to be released.

There are two phases in 2PL, an expanding phase and a shrinking one. In the expanding phase, the transaction is allowed only to acquire locks but not release them. In the shrinking phase, the transaction is permitted only to release locks but not acquire them. If these rules are obeyed, it can be formally proven that the protocol guarantees strict serializability. In practice, locks are only released when the transaction completes (aka strict 2PL). This ensures that data written by an uncommitted transaction X is locked until it's committed, preventing another transaction Y from reading it and consequently aborting if X is aborted (aka *cascading abort*), resulting in wasted work.

Unfortunately, with 2PL, it's possible for two or more transactions to *deadlock* and get stuck. For example, if transaction X is waiting for a lock that transaction Y holds, and transaction Y is waiting for a lock granted to transaction X, then the two transactions won't make any progress. A general approach to deal with deadlocks is to detect them after they occur and select a "victim" transaction to abort and restart to break the deadlock.

In contrast to a pessimistic protocol, an *optimistic* protocol optimistically executes a transaction without blocking based on the

[7] "Two-phase locking," https://en.wikipedia.org/wiki/Two-phase_locking

assumption that conflicts are rare and transactions are short-lived. *Optimistic concurrency control*[8](OCC) is arguably the best-known protocol in the space. In OCC, a transaction writes to a local workspace without modifying the actual data store. Then, when the transaction wants to commit, the data store compares the transaction's workspace to see whether it conflicts with the workspace of another running transaction. This is done by assigning each transaction a timestamp that determines its serializability order. If the validation succeeds, the content of the local workspace is copied to the data store. If the validation fails, the transaction is aborted and restarted.

It's worth pointing out that OCC uses *locks* to guarantee mutual exclusion on internal shared data structures. These *physical* locks are held for a short duration and are unrelated to the *logical* locks we discussed earlier in the context of 2PL. For example, during the validation phase, the data store has to acquire locks to access the workspaces of the running transactions to avoid race conditions. In the database world, these locks are also referred to as *latches* to distinguish them from logical locks.

Optimistic protocols avoid the overhead of pessimistic protocols, such as acquiring locks and managing deadlocks. As a result, these protocols are well suited for read-heavy workloads that rarely perform writes or workloads that perform writes that only occasionally conflict with each other. On the other hand, pessimistic protocols are more efficient for conflict-heavy workloads since they avoid wasting work.

Taking a step back, both the optimistic and pessimistic protocols discussed this far aren't optimal for read-only transactions. In 2PC, a read-only transaction might wait for a long time to acquire a shared lock. On the other hand, in OCC, a read-only transaction may be aborted because the value it read has been overwritten. Generally, the number of read-only transactions is much higher than the number of write transactions, so it would be ideal if a read-only transaction could never block or abort because of a con-

[8]"On Optimistic Methods for Concurrency Control," https://www.eecs.harvard.edu/~htk/publication/1981-tods-kung-robinson.pdf

flict with a write transaction.

Multi-version concurrency control[9] (MVCC) delivers on that premise by maintaining older versions of the data. Conceptually, when a transaction writes an object, the data store creates a new version of it. And when the transaction reads an object, it reads the newest version that existed when the transaction started. This mechanism allows a read-only transaction to read an immutable and consistent snapshot of the data store without blocking or aborting due to a conflict with a write transaction. However, for write transactions, MVCC falls back to one of the concurrency control protocols we discussed before (i.e., 2PL or OCC). Since generally most transactions are read-only, this approach delivers major performance improvements, which is why MVCC is the most widely used concurrency control scheme nowadays.

For example, when MVCC is combined with 2PL, a write transaction uses 2PL to access any objects it wants to read or write so that if another transaction tries to update any of them, it will block. When the transaction is ready to commit, the transaction manager gives it an unique *commit timestamp* TC_i, which is assigned to all new versions the transaction created. Because only a single transaction can commit at a time, this guarantees that once the transaction commits, a read-only transaction whose *start timestamp* TS_j is greater than or equal to the commit timestamp of the previous transaction ($TS_j \geq TC_i$), will see all changes applied by the previous transaction. This is a consequence of the protocol allowing read-only transactions to read only the newest committed version of an object that has a timestamp less than or equal TS_j. Thanks to this mechanism, a read-only transaction can read an immutable and consistent snapshot of the data store without blocking or aborting due to a conflict with a write transaction.

I have deliberately glossed over the details of how these concurrency control protocols are implemented, as it's unlikely you will have to implement them from scratch. But, the commercial data stores your applications depend on use the above protocols to iso-

[9] "Multi-version concurrency control," https://en.wikipedia.org/wiki/Multiversion_concurrency_control

late transactions, so you must have a basic grasp of their trade-offs. If you are interested in the details, I highly recommend Andy Pavlo's database systems course[10].

That said, there is a limited form of OCC at the level of individual objects that is widely used in distributed applications and that you should know how to implement. The protocol assigns a version number to each object, which is incremented every time the object is updated. A transaction can then read a value from a data store, do some local computation, and finally update the value conditional on the version of the object not having changed. This validation step can be performed atomically using a compare-and-swap operation, which is supported by many data stores.[11] For example, if a transaction reads version 42 of an object, it can later update the object only if the version hasn't changed. So if the version is the same, the object is updated and the version number is incremented to 43 (atomically). Otherwise, the transaction is aborted and restarted.

12.3 Atomicity

When executing a transaction, there are two possible outcomes: it either commits after completing all its operations or aborts due to a failure after executing some operations. When a transaction is aborted, the data store needs to guarantee that all the changes the transaction performed are undone (rolled back).

To guarantee atomicity (and also durability), the data store records all changes to a write-ahead log (WAL) persisted on disk before applying them. Each log entry records the identifier of the transaction making the change, the identifier of the object being modified, and both the old and new value of the object. Most of the time, the database doesn't read from this log at all. But if a transaction is aborted or the data store crashes, the log contains enough information to redo changes to ensure atomicity and durability and

[10] "CMU 15-445/645: Database Systems," https://15445.courses.cs.cmu.edu/fall2019/

[11] We have already seen an example of this when discussing leases in section 9.2.

undo changes in case of a failure during a transaction execution.[12]

Unfortunately, this WAL-based recovery mechanism only guarantees atomicity within a single data store. Going back to our original example of sending money from one bank account to another, suppose the two accounts belong to two different banks that use separate data stores. We can't just run two separate transactions to respectively withdraw and deposit the funds — if the second transaction fails, the system is left in an inconsistent state. What we want is the guarantee that either both transactions succeed and their changes are committed or they fail without any side effects.

12.3.1 Two-phase commit

Two-phase commit[13] (2PC) is a protocol used to implement atomic transaction commits across multiple processes. The protocol is split into two phases, *prepare* and *commit*. It assumes a process acts as *coordinator* and orchestrates the actions of the other processes, called *participants*. For example, the client application that initiates the transaction could act as the coordinator for the protocol.

When a coordinator wants to commit a transaction, it sends a *prepare* request asking the participants whether they are prepared to commit the transaction (see Figure 12.2). If all participants reply that they are ready to commit, the coordinator sends a *commit* request to all participants ordering them to do so. In contrast, if any process replies that it's unable to commit or doesn't respond promptly, the coordinator sends an *abort* request to all participants.

There are two points of no return in the protocol. If a participant replies to a prepare message that it's ready to commit, it will have to do so later, no matter what. The participant can't make progress from that point onward until it receives a message from the coordinator to either commit or abort the transaction. This means that

[12] The details of how this recovery mechanism works are outside the scope of this book, but you can learn more about it from any database textbook.

[13] "Two-phase commit protocol," https://en.wikipedia.org/wiki/Two-phase_commit_protocol

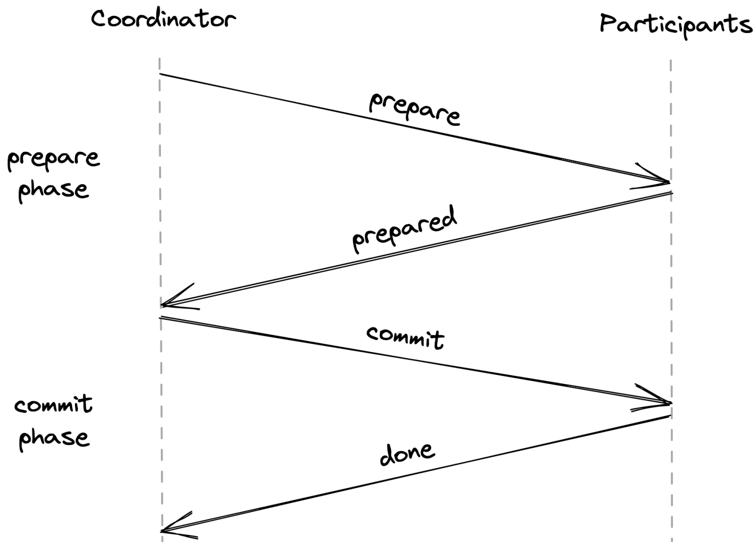

Figure 12.2: The two-phase commit protocol consists of a prepare and a commit phase.

if the coordinator crashes, the participant is stuck.

The other point of no return is when the coordinator decides to commit or abort the transaction after receiving a response to its prepare message from all participants. Once the coordinator makes the decision, it can't change its mind later and has to see the transaction through to being committed or aborted, no matter what. If a participant is temporarily down, the coordinator will keep retrying until the request eventually succeeds.

Two-phase commit has a mixed reputation[14]. It's slow since it requires multiple round trips for a transaction to complete, and if either the coordinator or a participant fails, then all processes part of the transactions are blocked until the failing process comes back online. On top of that, the participants need to implement the protocol; you can't just take two different data stores and expect them

[14] "It's Time to Move on from Two Phase Commit," http://dbmsmusings.blogspot.com/2019/01/its-time-to-move-on-from-two-phase.html

to play ball with each other.

If we are willing to increase the complexity of the protocol, we can make it more resilient to failures by replicating the state of each process involved in the transaction. For example, replicating the coordinator with a consensus protocol like Raft makes 2PC resilient to coordinator failures. Similarly, the participants can also be replicated.

As it turns out, atomically committing a transaction is a form of consensus, called *uniform consensus*, where all the processes have to agree on a value, even the faulty ones. In contrast, the general form of consensus introduced in section 10.2 only guarantees that all non-faulty processes agree on the proposed value. Therefore, uniform consensus is actually harder[15] than consensus. Nevertheless, as mentioned earlier, general consensus can be used to replicate the state of each process and make the overall protocol more robust to failures.

12.4 NewSQL

As a historical side note, the first versions of modern large-scale data stores that came out in the late 2000s used to be referred to as *NoSQL* stores since their core features were focused entirely on scalability and lacked the guarantees of traditional relational databases, such as ACID transactions. But in recent years, that has started to change as distributed data stores have continued to add features that only traditional databases offered.

Arguably one of the most successful implementations of a NewSQL data store is Google's Spanner[16]. At a high level, Spanner breaks data (key-value pairs) into partitions in order to scale. Each partition is replicated across a group of nodes in different data centers using a state machine replication protocol (Paxos).

[15] "Uniform consensus is harder than consensus," https://infoscience.epfl.ch/record/88273/files/CBS04.pdf?version=1

[16] "Spanner: Google's Globally-Distributed Database," https://static.googleusercontent.com/media/research.google.com/en//archive/spanner-osdi2012.pdf

In each replication group, there is one specific node that acts as the leader, which handles a client write transaction for that partition by first replicating it across a majority of the group and then applying it. The leader also serves as a lock manager and implements 2PL to isolate transactions modifying the partition from each other.

To support transactions that span multiple partitions, Spanner implements 2PC. A transaction is initiated by a client and coordinated by one of the group leaders of the partitions involved. All the other group leaders act as participants of the transaction (see Figure 12.3).

The coordinator logs the transaction's state into a local write-ahead log, which is replicated across its replication group. That way, if the coordinator crashes, another node in the replication group is elected as the leader and resumes coordinating the transaction. Similarly, each participant logs the transaction's state in its log, which is also replicated across its group. So if the participant fails, another node in the group takes over as the leader and resumes the transaction.

To guarantee isolation between transactions, Spanner uses MVCC combined with 2PL. Thanks to that, read-only transactions are lock-free and see a consistent snapshot of the data store. In contrast, write transactions use two-phase locking to create new versions of the objects they change to guarantee strict serializability.

If you recall the discussion about MVCC in section 12.2, you should know that each transaction is assigned a unique timestamp. While it's easy to do that on a single machine, it's a lot more challenging in a distributed setting since clocks aren't perfectly synchronized. Although we could use a centralized timestamp service that allocates unique timestamps to transactions, it would become a scalability bottleneck. To solve this problem, Spanner uses physical clocks that, rather than returning precise timestamps, return *uncertainty intervals* $[t_{earliest}, t_{latest}]$ that take into account the error boundary of the measurements, where $t_{earliest} \leq t_{real} \leq t_{latest}$. So although a node doesn't know the

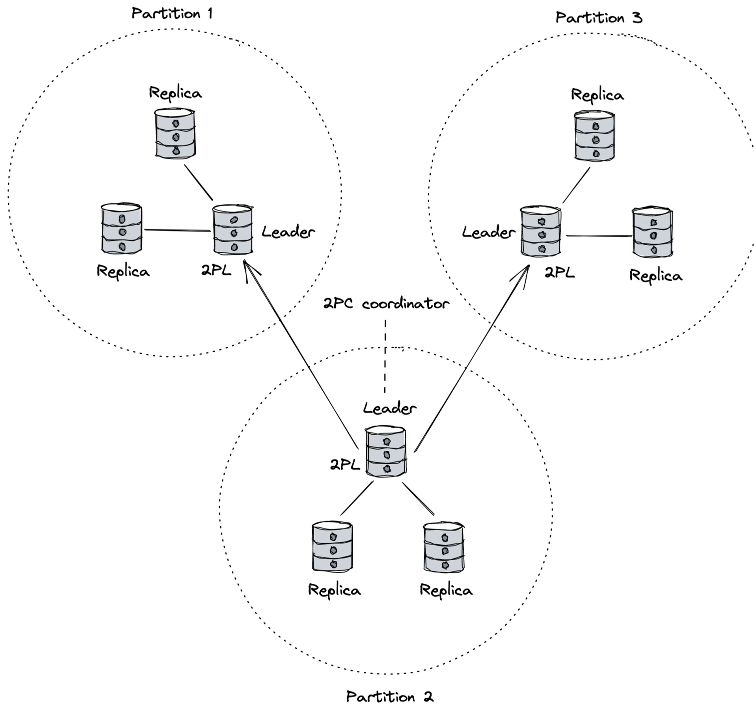

Figure 12.3: Spanner combines 2PC with 2PL and state machine replication. In this figure, there are three partitions and three replicas per partition.

current physical time t_{real}, it knows it's within the interval with a very high probability.

Conceptually, when a transaction wants to commit, it's assigned the t_{latest} timestamp of the interval returned by the transaction coordinator's clock. But before the transaction can commit and release the locks, it waits for a duration equal to the uncertainty period ($t_{latest} - t_{earliest}$). The waiting time guarantees that any transaction that starts after the previous transaction committed sees the changes applied by it. Of course, the challenge is to keep the uncertainty interval as small as possible in order for the transactions to be fast. Spanner does this by deploying GPS and atomic clocks in every data center and frequently synchronizing the quartz clocks

of the nodes with them.[17]. Other systems inspired by Spanner, like CockroachDB[18], take a different approach and rely instead on hybrid-logical clocks which are composed of a physical timestamp and a logical timestamp.[19]

[17] The clock uncertainty is generally less than 10ms.

[18] "CockroachDB," https://www.cockroachlabs.com/

[19] "Logical Physical Clocks and Consistent Snapshots in Globally Distributed Databases," https://cse.buffalo.edu/tech-reports/2014-04.pdf

Chapter 13

Asynchronous transactions

2PC is a synchronous blocking protocol — if the coordinator or any of the participants is slow or not not available, the transaction can't make progress. Because of its blocking nature, 2PC is generally combined with a blocking concurrency control protocol, like 2PL, to provide isolation. That means the participants are holding locks while waiting for the coordinator, blocking other transactions accessing the same objects from making progress.

The underlying assumptions of 2PC are that the coordinator and the participants are available and that the duration of the transaction is short-lived. While we can do something about the participants' availability by using state machine replication, we can't do much about transactions that, due to their nature, take a long time to execute, like hours or days. In this case, blocking just isn't an option. Additionally, if the participants belong to different organizations, the organizations might be unwilling to grant each other the power to block their systems to run transactions they don't control.

To solve this problem, we can look for solutions in the real world. For example, consider a fund transfer between two banks via a

cashier's check. First, the bank issuing the check deducts the funds from the source account. Then, the check is physically transported to the other bank and there it's finally deposited to the destination account. For the fund transfer to work, the check cannot be lost or deposited more than once. Since neither the source nor destination bank had to wait on each other while the transaction was in progress, the fund transfer via check is an asynchronous (non-blocking) atomic transaction. However, the price to pay for this is that the source and destination account are in an inconsistent state while the check is being transferred. So although asynchronous transactions are atomic, they are not isolated from each other.

Now, because a check is just a message, we can generalize this idea with the concept of *persistent* messages sent over the network, i.e., messages that that are guaranteed to be processed *exactly once*. In this chapter, we will discuss a few implementations of asynchrounous transactions based on this concept.

13.1 Outbox pattern

A common pattern[1] in modern applications is to replicate the same data to different data stores tailored to different use cases. For example, suppose we own a product catalog service backed by a relational database, and we decide to offer an advanced full-text search capability in its API. Although some relational databases offer a basic full-text search functionality, a dedicated service such as Elasticsearch[2] is required for more advanced use cases.

To integrate with the search service, the catalog service needs to update both the relational database and the search service when a new product is added, or an existing product is modified or deleted. The service could update the relational database first and then the search service, but if the service crashes before updating the search service, the system would be left in an inconsistent

[1] "Online Event Processing: Achieving consistency where distributed transactions have failed," https://queue.acm.org/detail.cfm?id=3321612

[2] "Elasticsearch: A distributed, RESTful search and analytics engine," https://www.elastic.co/elasticsearch/

state. So we need to wrap the two updates into a transaction somehow.

We could consider using 2PC, but while the relational database supports the X/Open XA[3] 2PC standard, the search service doesn't, which means we would have to implement the protocol for the search service somehow. We also don't want the catalog service to block if the search service is temporarily unavailable. Although we want the two data stores to be in sync, we can accept some temporary inconsistencies. So eventual consistency is acceptable for our use case.

We can solve this problem by having the catalog service send a persistent message to the search service whenever a product is added, modified or deleted. One way of implementing that is for a local transaction to append the message to a dedicated *outbox* table[4] when it makes a change to the product catalog. Because the relational database supports ACID transactions, the message is appended to the outbox table if and only if the local transaction commits and is not aborted.

The outbox table can then be monitored by a dedicated *relay process*. When the relay process discovers a new message, it sends the message to the destination, the search service. The relay process deletes the message from the table only when it receives an acknolowedgment that it was was delivered successfully. Unsurprisingly, it's possible for the same message to be delivered multiple times. For example, if the relay process crashes after sending the message but before removing it from the table, it will resend the message when it restarts. To guarantee that the destination processes the message only once, an idempotency key is assigned to it so that the message can be deduplicated (we discussed this in chapter 5.7).

In practice, the relay process doesn't send messages directly to the

[3]"Distributed Transaction Processing: The XA Specification," https://pubs.opengroup.org/onlinepubs/009680699/toc.pdf

[4]"Pattern: Transactional outbox," https://microservices.io/patterns/data/transactional-outbox.html

destination. Instead, it forwards messages to a message channel[5], like Kafka[6] or Azure Event Hubs[7], responsible for delivering them to one or more destinations in the same order as they were appended. Later in chapter 23, we will discuss message channels in more detail.

If you squint a little, you will see that what we have just implemented here is conceptually similar to state machine replication, where the state is represented by the products in the catalog, and the replication happens through a log of operations (the outbox table).

13.2 Sagas

Now suppose we own a travel booking service. To book a trip, the travel service has to atomically book a flight through a dedicated service and a hotel through another. However, either of these services can fail their respective request. If one booking succeeds, but the other fails, then the former needs to be canceled to guarantee atomicity. Hence, booking a trip requires multiple steps to complete, some of which are only required in case of failure. For that reason, we can't use the simple solution presented earlier.

The *Saga*[8] pattern provides a solution to this problem. A saga is a distributed transaction composed of a set of local transactions $T_1, T_2, ..., T_n$, where T_i has a corresponding compensating local transaction C_i used to undo its changes. The saga guarantees that either all local transactions succeed, or, in case of failure, the compensating local transactions undo the partial execution of the transaction altogether. This guarantees the atomicity of the protocol; either all local transactions succeed, or none of them do.

[5]For example, Debezium is an open source relay service that does this, see https://debezium.io/.

[6]"Apache Kafka: An open-source distributed event streaming platform," https://kafka.apache.org

[7]"Azure Event Hubs: A fully managed, real-time data ingestion service," https://azure.microsoft.com/en-gb/services/event-hubs/

[8]"Sagas," https://www.cs.cornell.edu/andru/cs711/2002fa/reading/sagas.pdf

Another way to think about sagas is that every local transaction T_i assumes all the other local transactions will succeed. It's a guess, and it's likely to be a good one, but still a guess at the end of the day. So when the guess is wrong, a mistake has been made, and an "apology[9]" needs to be issued in the form of compensating transactions C_i. This is similar to what happens in the real world when, e.g., a flight is overbooked.

A saga can be implemented with an orchestrator, i.e., the transaction coordinator, that manages the execution of the local transactions across the processes involved, i.e., the transaction's participants. In our example, the travel booking service is the transaction's coordinator, while the flight and hotel booking services are the transaction's participants. The saga is composed of three local transactions: T_1 books a flight, T_2 books a hotel, and C_1 cancels the flight booked with T_1.

At a high level, the saga can be implemented with the *workflow*[10] depicted in Figure 13.1:

1. The coordinator initiates the transaction by sending a booking request (T_1) to the flight service. If the booking fails, no harm is done, and the coordinator marks the transaction as aborted.
2. If the flight booking succeeds, the coordinator sends a booking request (T_2) to the hotel service. If the request succeeds, the transaction is marked as successful, and we are all done.
3. If the hotel booking fails, the transaction needs to be aborted. The coordinator sends a cancellation request (C_1) to the flight service to cancel the previously booked flight. Without the cancellation, the transaction would be left in an inconsistent state, which would break its atomicity guarantee.

The coordinator can communicate asynchronously with the participants via message channels to tolerate temporary failures. As the transaction requires multiple steps to succeed, and the coordinator

[9] "Building on Quicksand," https://dsf.berkeley.edu/cs286/papers/quicksand-cidr2009.pdf

[10] "Clarifying the Saga pattern," http://web.archive.org/web/20161205130022/http://kellabyte.com:80/2012/05/30/clarifying-the-saga-pattern

can fail at any time, it needs to persist the state of the transaction as it advances. By modeling the transaction as a state machine, the coordinator can durably checkpoint its state to a data store as it transitions from one state to the next. This ensures that if the coordinator crashes and restarts, or another process is elected as the coordinator, it can resume the transaction from where it left off by reading the last checkpoint.

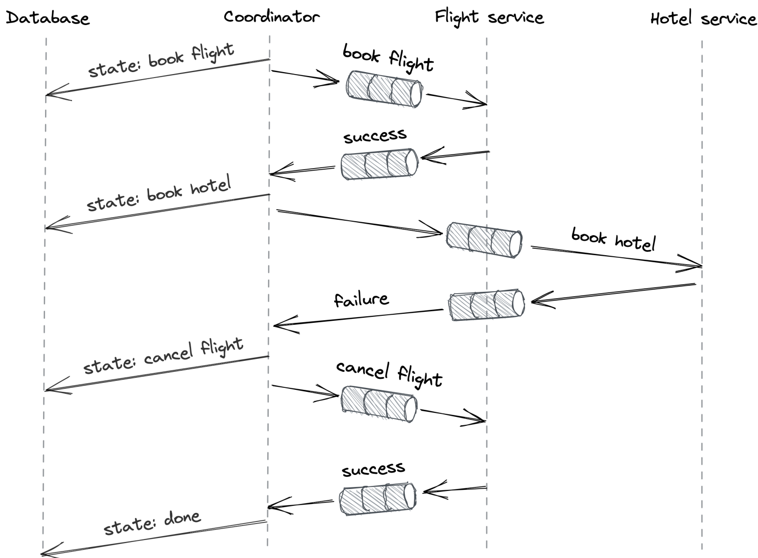

Figure 13.1: A workflow implementing an asynchronous transaction

There is a caveat, though; if the coordinator crashes after sending a request but before backing up its state, it will send the same request again when it comes back online. Similarly, if sending a request times out, the coordinator will have to retry it, causing the message to appear twice at the receiving end. Hence, the participants have to de-duplicate the messages they receive to make them idempotent.

In practice, you don't need to build orchestration engines from scratch to implement such workflows, since cloud compute ser-

vices such as AWS Step Functions[11] or Azure Durable Functions[12] make it easy to create managed workflows.

13.3 Isolation

We started our journey into asynchronous transactions as a way to work around the blocking nature of 2PC. But to do that, we had to sacrifice the isolation guarantee that traditional ACID transactions provide. As it turns out, we can work around the lack of isolation as well. For example, one way to do that is by using *semantic locks*[13]. The idea is that any data the saga modifies is marked with a *dirty flag*, which is only cleared at the end of the transaction. Another transaction trying to access a dirty record can either fail and roll back its changes or wait until the dirty flag is cleared.

[11] "AWS Step Functions," https://aws.amazon.com/step-functions/

[12] "Azure Durable Functions documentation ," https://docs.microsoft.com/en-us/azure/azure-functions/durable/

[13] "Semantic ACID properties in multidatabases using remote procedure calls and update propagations," https://dl.acm.org/doi/10.5555/284472.284478

Summary

If you pick any textbook about distributed systems or database systems, I guarantee you will find entire chapters dedicated to the topics discussed in this part. In fact, you can find entire books written about them! Although you could argue that it's unlikely you will ever have to implement core distributed algorithms such as state machine replication from scratch, I feel it's important to have seen these at least once as it will make you a better "user" of the abstractions they provide.

There are two crucial insights I would like you to take away from this part. One is that failures are unavoidable in distributed systems, and the other is that coordination is expensive.

By now, you should have realized that what makes the coordination algorithms we discussed so complex is that they must tolerate failures. Take Raft, for example. Imagine how much simpler the implementation would be if the network were reliable and processes couldn't crash. If you were surprised by one or more edge cases in the design of Raft or chain replication, you are not alone. This stuff is hard! Fault tolerance plays a big role at any level of the stack, which is why I dedicated Part IV to it.

The other important takeaway is that coordination adds complexity and impacts scalability and performance. Hence, you should strive to reduce coordination when possible by:

- keeping coordination off the critical path, as chain replication does;

- proceeding without coordination and “apologize” when an inconsistency is detected, as sagas do;
- using protocols that guarantee some form of consistency without coordination, like CRDTs.

Part III

Scalability

Introduction

> *"Treat servers like cattle, not pets."*
>
> – Bill Baker

Over the last few decades, the number of people with access to the internet has steadily climbed. In 1996, only 1% of people worldwide had access to the internet, while today, it's over 65%. In turn, this has increased the total addressable market of online businesses and led to the need for scalable systems that can handle millions of concurrent users.

For an application to scale, it must run without performance degradations as load increases. And as mentioned in chapter 1, the only long-term solution for increasing the application's capacity is to architect it so that it can scale horizontally.

In this part, we will walk through the journey of scaling a simple CRUD web application called *Cruder*. *Cruder* is comprised of a single-page JavaScript application that communicates with an application server through a RESTful HTTP API. The server uses the local disk to store large files, like images and videos, and a relational database to persist the application's state. Both the database and the application server are hosted on the same machine, which is managed by a compute platform like AWS EC2. Also, the server's public IP address is advertised by a managed DNS service, like AWS Route 53[14].

[14] "Amazon Route 53," https://aws.amazon.com/route53/

Users interact with *Cruder* through their browsers. Typically, a browser issues a DNS request to resolve the domain name to an IP address (if it doesn't have it cached already), opens a TLS connection with the server, and sends its first HTTP *GET* request to it (see Figure 13.2).

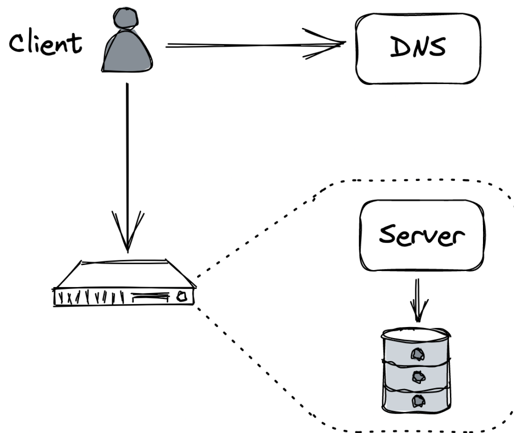

Figure 13.2: Cruder's architecture

Although this architecture is good enough for a proof of concept, it's not scalable or fault-tolerant. Of course, not all applications need to be highly available and scalable, but since you are reading this book to learn how such applications are built, we can assume that the application's backend will eventually need to serve millions of requests per second.

And so, as the number of requests grows, the application server will require more resources (e.g., CPU, memory, disk, network) and eventually reach its capacity, and its performance will start to degrade. Similarly, as the database stores more data and serves more queries, it will inevitably slow down as it competes for resources with the application server.

The simplest and quickest way to increase the capacity is to *scale up* the machine hosting the application. For example, we could:

- increase the number of threads capable of running simulta-

neously by provisioning more processors or cores,
- increase disk throughput by provisioning more disks (RAID),
- increase network throughput by provisioning more NICs,
- reduce random disk access latency by provisioning solid-state disks (SSD),
- or reduce page faults by provisioning more memory.

The caveat is that the application needs to be able to leverage the additional hardware at its disposal. For example, adding more cores to a single-threaded application will not make much difference. More importantly, when we've maxed out on the hardware front, the application will eventually hit a hard physical limit that we can't overcome no matter how much money we are willing to throw at it.

The alternative to scaling up is to *scale out* by distributing the application across multiple nodes. Although this makes the application more complex, eventually it will pay off. For example, we can move the database to a dedicated machine as a first step. By doing that, we have increased the capacity of both the server and the database since they no longer have to compete for resources. This is an example of a more general pattern called *functional decomposition*: breaking down an application into separate components, each with its own well-defined responsibility (see Figure 13.3).

As we will repeatedly see in the following chapters, there are other two general patterns that we can exploit (and combine) to build scalable applications: splitting data into partitions and distributing them among nodes (*partitioning*) and replicating functionality or data across nodes, also known as horizontal scaling (*replication*). In the following chapters, we will explore techniques based on those patterns for further increasing *Cruder*'s capacity, which require increasingly more effort to exploit.

Chapter 14 discusses the use of client-side caching to reduce the number of requests hitting the application.

Chapter 15 describes using a content delivery network (CDN), a geographically distributed network of managed reverse proxies,

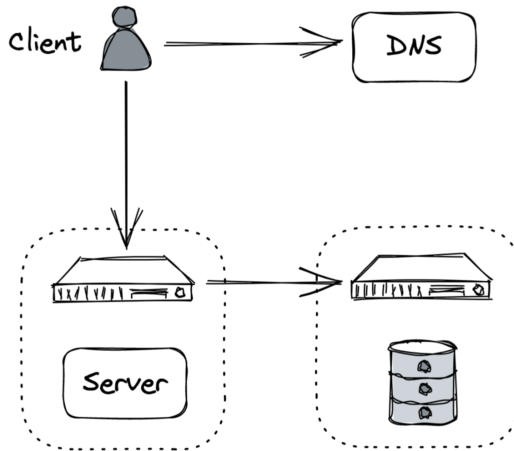

Figure 13.3: Moving the database to its own dedicated machine is an example of functional decomposition

to further reduce the number of requests the application needs to handle.

Chapter 16 dives into partitioning, a technique used by CDNs, and pretty much any distributed data store, to handle large volumes of data. The chapter explores different partitioning strategies, such as range and hash partitioning, and the challenges that partitioning introduces.

Chapter 17 discusses the benefits of offloading the storage of large static files, such as images and videos, to a managed file store. It then describes the architecture of Azure Storage, a highly available and scalable file store.

Chapter 18 talks about how to increase the application's capacity by load-balancing requests across a pool of servers. The chapter starts with a simple approach based on DNS and then explores more flexible solutions that operate at the transport and application layers of the network stack.

Chapter 19 describes how to scale out the application's relational database using replication and partitioning and the challenges that

come with it. It then introduces NoSQL data stores as a solution to these challenges and recounts their evolution since their initial adoption in the industry.

Chapter 20 takes a stab at discussing caching from a more general point of view by diving into the benefits and pitfalls of putting a cache in front of the application's data store. Although caching is a deceptively simple technique, it can create subtle consistency and operational issues that are all too easy to dismiss.

Chapter 21 talks about scaling the development of the application across multiple teams by decomposing it into independently deployable services. Next, it introduces the concept of an API gateway as a means for external clients to communicate with the backend after it has been decomposed into separated services.

Chapter 22 describes the benefits of separating the serving of client requests (*data plane*) from the management of the system's metadata and configuration (*control plane*), which is a common pattern in large-scale systems.

Chapter 23 explores the use of asynchronous messaging channels to decouple the communication between services, allowing two services to communicate even if one of them is temporarily unavailable. Messaging offers many other benefits, which we will explore in the chapter along with its best practices and pitfalls.

Chapter 14

HTTP caching

Cruder's application server handles both static and dynamic resources at this point. A static resource contains data that usually doesn't change from one request to another, like a JavaScript or CSS file. Instead, a dynamic resource is generated by the server on the fly, like a JSON document containing the user's profile.

Since a static resource doesn't change often, it can be cached. The idea is to have a client (i.e., a browser) cache the resource for some time so that the next access doesn't require a network call, reducing the load on the server and the response time.[1]

Let's see how client-side HTTP caching works in practice with a concrete example. HTTP caching is limited to safe request methods that don't alter the server's state, like GET or HEAD. Suppose a client issues a GET request for a resource it hasn't accessed before. The local cache intercepts the request, and if it can't find the resource locally, it will go and fetch it from the origin server on behalf of the client.

The server uses specific HTTP response headers[2] to let clients know that a resource is cachable. So when the server responds

[1] This is an example of the replication pattern in action.

[2] "HTTP caching," https://developer.mozilla.org/en-US/docs/Web/HTTP/Caching

with the resource, it adds a *Cache-Control* header that defines for how long to cache the resource (TTL) and an *ETag* header with a version identifier. Finally, when the cache receives the response, it stores the resource locally and returns it to the client (see Figure 14.1).

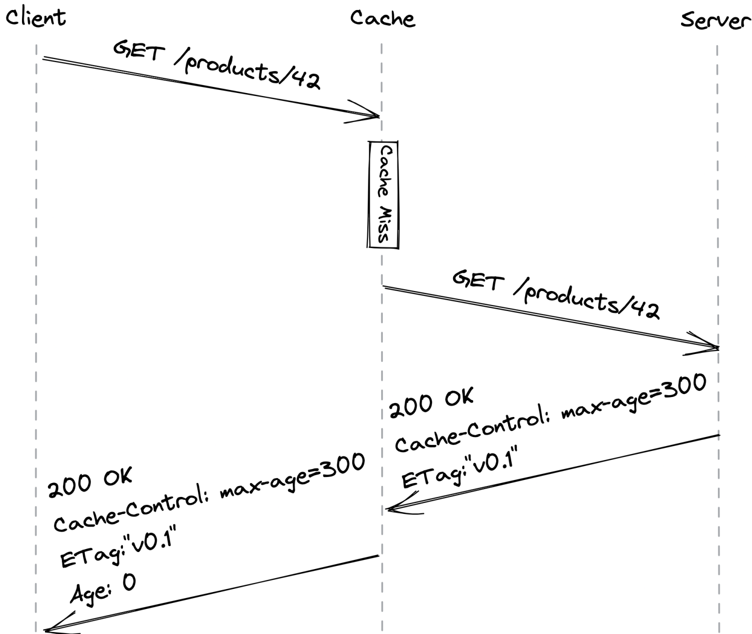

Figure 14.1: A client accessing a resource for the first time (the Age header contains the time in seconds the object was in the cache)

Now, suppose some time passes and the client tries to access the resource again. The cache first checks whether the resource hasn't expired yet, i.e., whether it's *fresh*. If so, the cache immediately returns it. However, even if the resource hasn't expired from the client's point of view, the server may have updated it in the meantime. That means reads are not strongly consistent, but we will safely assume that this is an acceptable tradeoff for our application.

If the resource has expired, it's considered *stale*. In this case, the

cache sends a GET request to the server with a conditional header (like *If-None-Match*) containing the version identifier of the stale resource to check whether a newer version is available. If there is, the server returns the updated resource; if not, the server replies with a *304 Not Modified* status code (see Figure 14.2).

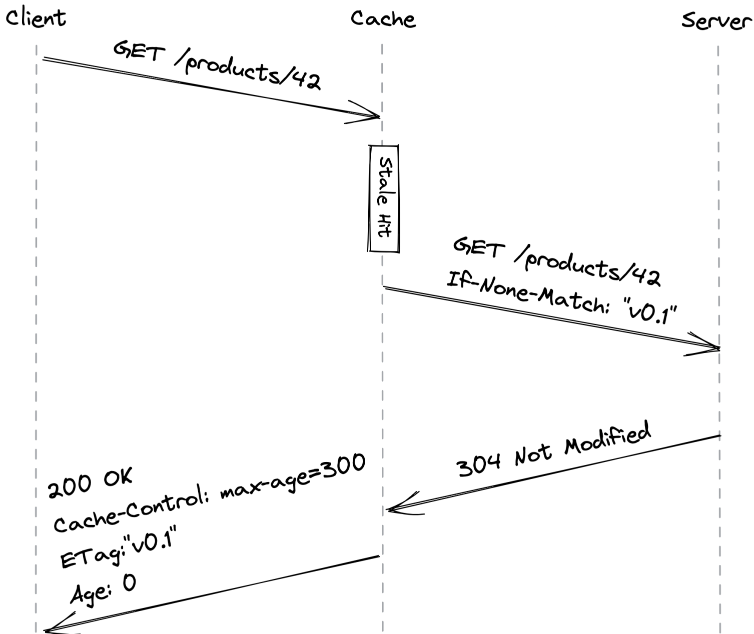

Figure 14.2: A client accessing a stale resource

Ideally, a static resource should be immutable so that clients can cache it "forever," which corresponds to a maximum length of a year according to the HTTP specification. We can still modify a static resource if needed by creating a new one with a different URL so that clients will be forced to fetch it from the server.

Another advantage of treating static resources as immutable is that we can update multiple related resources atomically. For example, if we publish a new version of the application's website, the updated HTML index file is going to reference the new URLs for the JavaScript and CSS bundles. Thus, a client will see either the old

version of the website or the new one depending on which index file it has read, but never a mix of the two that uses, e.g., the old JavaScript bundle with the new CSS one.

Another way of thinking about HTTP caching is that we treat the read path (*GET*) differently from the write path (*POST, PUT, DELETE*) because we expect the number of reads to be several orders of magnitude higher than the number of writes. This is a common pattern referred to as the *Command Query Responsibility Segregation*[3] (CQRS) pattern.[4]

To summarize, allowing clients to cache static resources has reduced the load on our server, and all we had to do was to play with some HTTP headers! We can take caching one step further by introducing a server-side HTTP cache with a reverse proxy.

14.1 Reverse proxies

A *reverse proxy* is a server-side proxy that intercepts all communications with clients. Since the proxy is indistinguishable from the actual server, clients are unaware that they are communicating through an intermediary (see Figure 14.3).

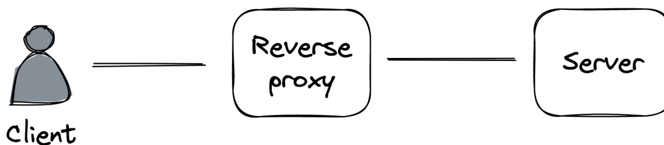

Figure 14.3: A reverse proxy acts as an intermediary between the clients and the servers.

A common use case for a reverse proxy is to cache static resources returned by the server. Since the cache is shared among the clients, it will decrease the load of the server a lot more than any client-side cache ever could.

[3] "CQRS," https://martinfowler.com/bliki/CQRS.html
[4] This is another instance of functional decomposition.

Because a reverse proxy is a middleman, it can be used for much more than just caching. For example, it can:

- authenticate requests on behalf of the server;
- compress a response before returning it to the client to speed up the transmission;
- rate-limit requests coming from specific IPs or users to protect the server from being overwhelmed;
- load-balance requests across multiple servers to handle more load.

We will explore some of these use cases in the next chapters. NGINX[5] and HAProxy[6] are widely-used reverse proxies that we could leverage to build a server-side cache. However, many use cases for reverse proxies have been commoditized by managed services. So, rather than building out a server-side cache, we could just leverage a *Content Delivery Network* (CDN).

[5]"NGINX," https://www.nginx.com/
[6]"HAProxy," https://www.haproxy.com/

Chapter 15

Content delivery networks

A CDN is an overlay network of geographically distributed caching servers (reverse proxies) architected around the design limitations of the network protocols that run the internet.

When using a CDN, clients hit URLs that resolve to caching servers that belong to the CDN. When a CDN server receives a request, it checks whether the requested resource is cached locally. If not, the CDN server transparently fetches it from the *origin server* (i.e., our application server) using the original URL, caches the response locally, and returns it to the client. AWS CloudFront[1] and Akamai[2] are examples of well-known CDN services.

15.1 Overlay network

You would think that the main benefit of a CDN is caching, but it's actually the underlying network substrate. The public internet is composed of thousands of networks, and its core routing protocol, BGP, was not designed with performance in mind. It primarily

[1] "Amazon CloudFront," https://aws.amazon.com/cloudfront/
[2] "Akamai," https://www.akamai.com/

uses the number of hops to cost how expensive a path is with respect to another, without considering their latencies or congestion.

As the name implies, a CDN is a network. More specifically, an overlay network[3] built on top of the internet that exploits a variety of techniques to reduce the response time of network requests and increase the bandwidth of data transfers.

When we first discussed TCP in chapter 2, we talked about the importance of minimizing the latency between a client and a server. No matter how fast the server is, if the client is located on the other side of the world from it, the response time is going to be over 100 ms just because of the network latency, which is physically limited by the speed of light. Not to mention the increased error rate when sending data across the public internet over long distances.

This is why CDN clusters are placed in multiple geographical locations to be closer to clients. But how do clients know which cluster is closest to them? One way is via *global DNS load balancing*[4]: an extension to DNS that considers the location of the client inferred from its IP, and returns a list of the geographically closest clusters taking into account also the network congestion and the clusters' health.

CDN servers are also placed at *internet exchange points*, where ISPs connect to each other. That way, virtually the entire communication from the origin server to the clients flows over network links that are part of the CDN, and the brief hops on either end have low latencies due to their short distance.

The routing algorithms of the overlay network are optimized to select paths with reduced latencies and congestion, based on continuously updated data about the health of the network. Additionally, TCP optimizations are exploited where possible, such as using pools of persistent connections between servers

[3] "The Akamai Network: A Platform for HighPerformance Internet Applications," https://groups.cs.umass.edu/ramesh/wp-content/uploads/sites/3/2019/12/The-akamai-network-a-platform-for-high-performance-internet-applications.pdf

[4] "Load Balancing at the Frontend," https://landing.google.com/sre/sre-book/chapters/load-balancing-frontend/

to avoid the overhead of setting up new connections and using optimal TCP window sizes to maximize the effective bandwidth (see Figure 15.1).

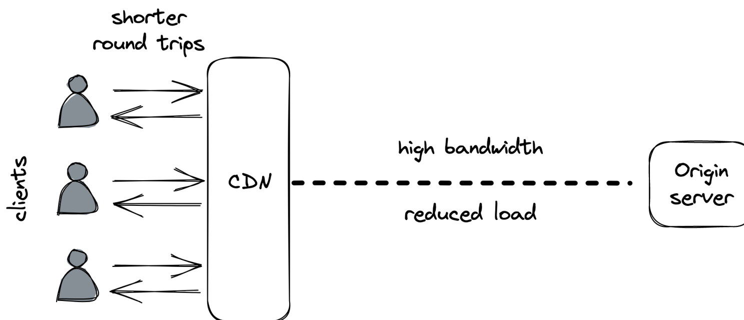

Figure 15.1: A CDN reduces the round trip time of network calls for clients and the load for the origin server.

The overlay network can also be used to speed up the delivery of dynamic resources that cannot be cached. In this capacity, the CDN becomes the frontend for the application, shielding it against distributed denial-of-service (DDoS) attacks[5].

15.2 Caching

A CDN can have multiple content caching layers. The top layer is made of edge clusters deployed at different geographical locations, as mentioned earlier. But infrequently accessed content might not be available at the edge, in which case the edge servers must fetch it from the origin server. Thanks to the overlay network, the content can be fetched more efficiently and reliably than what the public internet would allow.

There is a tradeoff between the number of edge clusters and the cache *hit ratio*[6], i.e., the likelihood of finding an object in the cache.

[5] "Denial-of-service attack," https://en.wikipedia.org/wiki/Denial-of-service_attack

[6] A cache hit occurs when the requested data can be found in the cache, while a cache miss occurs when it cannot.

The higher the number of edge clusters, the more geographically dispersed clients they can serve, but the lower the cache hit ratio will be, and consequently, the higher the load on the origin server. To alleviate this issue, the CDN can have one or more intermediary caching clusters deployed in a smaller number of geographical locations, which cache a larger fraction of the original content.

Within a CDN cluster, the content is partitioned among multiple servers so that each one serves only a specific subset of it; this is necessary as no single server would be able to hold all the data. Because data partitioning is a core scalability pattern, we will take a closer look at it in the next chapter.

Chapter 16

Partitioning

When an application's data keeps growing in size, its volume will eventually become large enough that it can't fit on a single machine. To work around that, it needs to be split into partitions, or *shards*, small enough to fit into individual nodes. As an additional benefit, the system's capacity for handling requests increases as well, since the load of accessing the data is spread over more nodes.

When a client sends a request to a partitioned system, the request needs to be routed to the node(s) responsible for it. A gateway service (i.e., a reverse proxy) is usually in charge of this, knowing how the data is mapped to partitions and nodes (see Figure 16.1). This mapping is generally maintained by a fault-tolerant coordination service, like etcd or Zookeeper.

Partitioning is not a free lunch since it introduces a fair amount of complexity:

- A gateway service is required to route requests to the right nodes.
- To roll up data across partitions, it needs to be pulled from multiple partitions and aggregated (e.g., think of the complexity of implementing a "group by" operation across partitions).

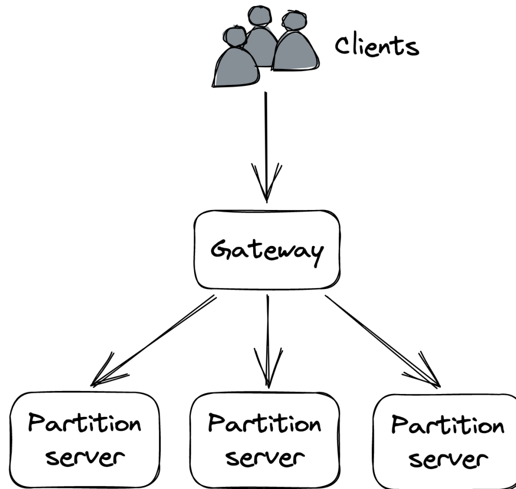

Figure 16.1: A partitioned application with a gateway that routes requests to partitions

- Transactions are required to atomically update data that spans multiple partitions, limiting scalability.
- If a partition is accessed much more frequently than others, the system's ability to scale is limited.
- Adding or removing partitions at run time becomes challenging, since it requires moving data across nodes.

It's no coincidence we are talking about partitioning right after discussing CDNs. A cache lends itself well to partitioning as it avoids most of this complexity. For example, it generally doesn't need to atomically update data across partitions or perform aggregations spanning multiple partitions.

Now that we have an idea of what partitioning is and why it's useful, let's discuss how data, in the form of key-value pairs, can be mapped to partitions. At a high level, there are two main ways of doing that, referred to as *range partitioning* and *hash partitioning*. An important prerequisite for both is that the number of possible keys is very large; for example, a boolean key with only two possible values is not suited for partitioning since it allows for at most two

partitions.

16.1 Range partitioning

Range partitioning splits the data by key range into lexicographically sorted partitions, as shown in Figure 16.2. To make range scans fast, each partition is generally stored in sorted order on disk.

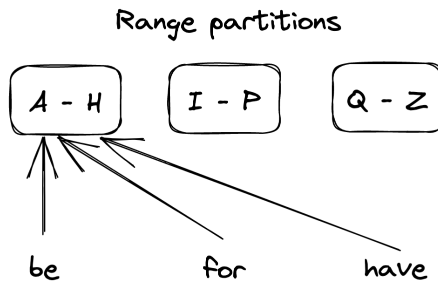

Figure 16.2: A range-partitioned dataset

The first challenge with range partitioning is picking the partition boundaries. For example, splitting the key range evenly makes sense if the distribution of keys is more or less uniform. If not, like in the English dictionary, the partitions will be unbalanced and some will have a lot more entries than others.

Another issue is that some access patterns can lead to *hotspots*, which affect performance. For example, if the data is range partitioned by date, all requests for the current day will be served by a single node. There are ways around that, like adding a random prefix to the partition keys, but there is a price to pay in terms of increased complexity.

When the size of the data or the number of requests becomes too large, the number of nodes needs to be increased to sustain the increase in load. Similarly, if the data shrinks and/or the number of requests drops, the number of nodes should be reduced to decrease costs. The process of adding and removing nodes to balance

the system's load is called *rebalancing*. Rebalancing has to be implemented in a way that minimizes disruption to the system, which needs to continue to serve requests. Hence, the amount of data transferred when rebalancing partitions should be minimized.

One solution is to create a lot more partitions than necessary when the system is first initialized and assign multiple partitions to each node. This approach is also called *static partitioning* since the number of partitions doesn't change over time. When a new node joins, some partitions move from the existing nodes to the new one so that the store is always in a balanced state. The drawback of this approach is that the number of partitions is fixed and can't be easily changed. Getting the number of partitions right is hard — too many partitions add overhead and decrease performance, while too few partitions limit scalability. Also, some partitions might end up being accessed much more than others, creating hotspots.

The alternative is to create partitions on demand, also referred to as *dynamic partitioning*. The system starts with a single partition, and when it grows above a certain size or becomes too hot, the partition is split into two sub-partitions, each containing approximately half of the data, and one of the sub-partitions is transferred to a new node. Similarly, when two adjacent partitions become small or "cold" enough, they can be merged into a single one.

16.2 Hash partitioning

Let's take a look at an alternative way of mapping data to partitions. The idea is to use a hash function that deterministically maps a key (string) to a seemingly random number (a hash) within a certain range (e.g., 0 and $2^{64} - 1$). This guarantees that the keys' hashes are distributed uniformly across the range.

Next, we assign a subset of the hashes to each partition, as shown in Figure 16.3. For example, one way of doing that is by taking the modulo of a hash, i.e., *hash(key) mod N*, where *N* is the number of partitions.

Although this approach ensures that the partitions contain more

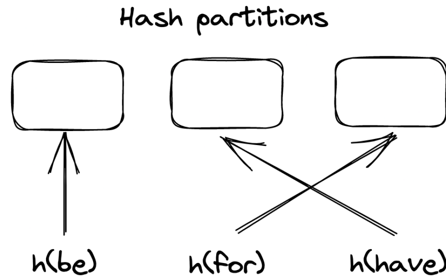

Figure 16.3: A hash-partitioned dataset

or less the same number of entries, it doesn't eliminate hotspots if the *access pattern* is not uniform. For example, if a single key is accessed significantly more often than others, the node hosting the partition it belongs to could become overloaded. In this case, the partition needs to be split further by increasing the total number of partitions. Alternatively, the key needs to be split into sub-keys by, e.g., prepending a random prefix.

Assigning hashes to partitions via the modulo operator can become problematic when a new partition is added, as most keys have to be moved (or shuffled) to a different partition because their assignment changed. Shuffling data is extremely expensive as it consumes network bandwidth and other resources. Ideally, if a partition is added, only $\frac{K}{N}$ keys should be shuffled around, where K is the number of keys and N is the number of partitions. One widely used hashing strategy with that property is consistent hashing.

With *consistent hashing*[1], a hash function randomly maps both the partition identifiers and keys onto a circle, and each key is assigned to the closest partition that appears on the circle in clockwise order (see Figure 16.4).

Now, when a new partition is added, only the keys that now map

[1] "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," https://www.cs.princeton.edu/courses/archive/fall09/cos518/papers/chash.pdf

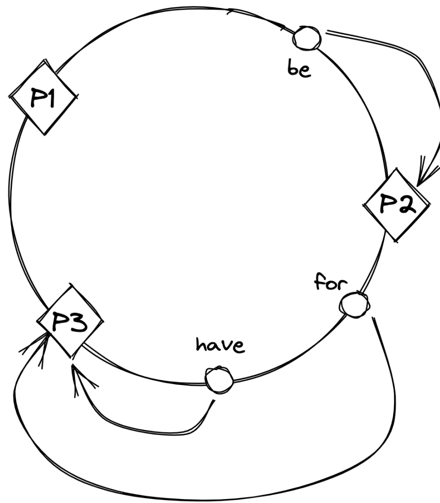

Figure 16.4: With consistent hashing, partition identifiers and keys are randomly distributed around a circle, and each key is assigned to the next partition that appears on the circle in clockwise order.

to it on the circle need to be reassigned, as shown in Figure 16.5.

The main drawback of hash partitioning compared to range partitioning is that the sort order over the partitions is lost, which is required to efficiently scan all the data in order. However, the data within an individual partition can still be sorted based on a secondary key.

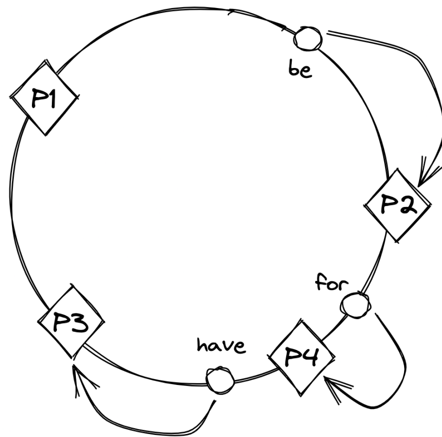

Figure 16.5: After partition P4 is added, the key 'for' is reassigned to P4, but the assignment of the other keys doesn't change.

Chapter 17

File storage

Using a CDN has significantly reduced the number of requests hitting *Cruder*'s application server. But there are only so many images, videos, etc., the server can store on its local disk(s) before running out of space. To work around this limit, we can use a managed file store, like AWS S3[1] or Azure Blob Storage[2], to store large static files. Managed file stores are scalable, highly available, and offer strong durability guarantees. A file uploaded to a managed store can be configured to allow access to anyone who knows its URL, which means we can point the CDN straight at it. This allows us to completely offload the storage and serving of static resources to managed services.

17.1 Blob storage architecture

Because distributed file stores are such a crucial component of modern applications, it's useful to have an idea of how they work underneath. In this chapter, we will dive into the architecture of

[1] "Amazon Simple Storage Service," https://aws.amazon.com/s3/

[2] "Azure Blob Storage," https://azure.microsoft.com/en-us/services/storage/blobs/#overview

Azure Storage[3] (AS), a scalable cloud storage system that provides strong consistency. AS supports file, queue, and table abstractions, but for simplicity, our discussion will focus exclusively on the file abstraction, also referred to as the blob store.

AS is composed of storage clusters distributed across multiple regions worldwide. A *storage cluster* is composed of multiple racks of nodes, where each rack is built out as a separate unit with redundant networking and power.

At a high level, AS exposes a global namespace based on domain names that are composed of two parts: an account name and a file name. The two names together form a unique URL that points to a specific file, e.g., *https://ACCOUNT_NAME.blob.core.windows.net/FILE_NAME*. The customer configures the account name, and the AS DNS server uses it to identify the storage cluster where the data is stored. The cluster uses the file name to locate the node responsible for the data.

A central *location service* acts as the global *control plane* in charge of creating new accounts and allocating them to clusters, and also moving them from one cluster to another for better load distribution. For example, when a customer wants to create a new account in a specific region, the location service:

- chooses a suitable cluster to which to allocate the account based on load information;
- updates the configuration of the cluster to start accepting requests for the new account;
- and creates a new DNS record that maps the account name to the cluster's public IP address.

From an architectural point of view, a storage cluster is composed of three layers: a stream layer, a partition layer, and a front-end layer (see Figure 17.1).

The *stream layer* implements a distributed append-only file system in which the data is stored in so-called streams. Internally, a *stream*

[3] "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency," https://sigops.org/s/conferences/sosp/2011/current/2011-Cascais/printable/11-calder.pdf

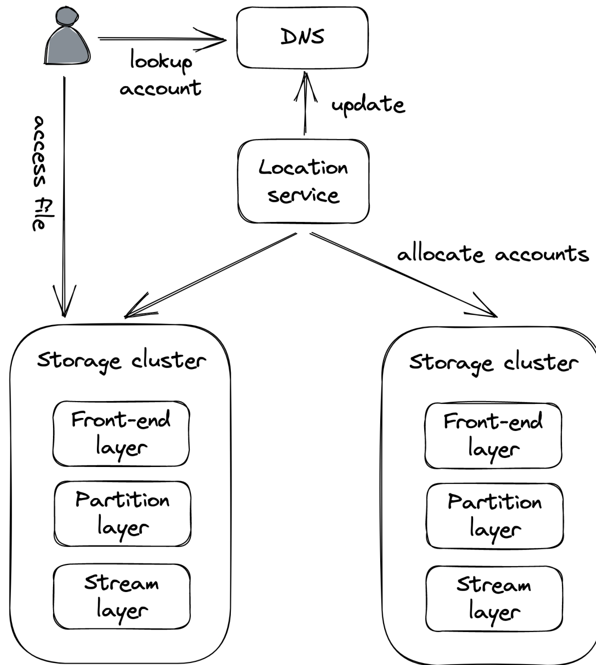

Figure 17.1: A high-level view of Azure Storage's architecture

is represented as a sequence of *extents*, where the extent is the unit of replication. Writes to extents are replicated synchronously using chain replication[4].

The *stream manager* is the control plane responsible for assigning an extent to a chain of storage servers in the cluster. When the manager is asked to allocate a new extent, it replies with the list of storage servers that hold a copy of the newly created extent (see Figure 17.2). The client caches this information and uses it to send future writes to the primary server. The stream manager is also responsible for handling unavailable or faulty extent replicas by creating new ones and reconfiguring the replication chains they are part of.

The *partition layer* is where high-level file operations are translated

[4] we discussed chain replication in section 10.4

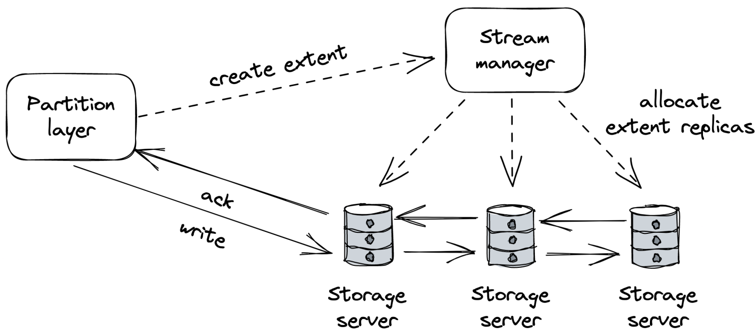

Figure 17.2: The stream layer uses chain replication to replicate extents across storage servers.

to low-level stream operations. Within this layer, the *partition manager* (yet another control plane) manages a large index of all files stored in the cluster. Each entry in the index contains metadata such as account and file name and a pointer to the actual data in the stream service (list of extent plus offset and length). The partition manager range-partitions the index and maps each partition to a partition server. The partition manager is also responsible for load-balancing partitions across servers, splitting partitions when they become too hot, and merging cold ones (see Figure 17.3).

The partition layer also asynchronously replicates accounts across clusters in the background. This functionality is used to migrate accounts from one cluster to another for load-balancing purposes and disaster recovery.

Finally, the *front-end service* (a reverse proxy) is a stateless service that authenticates requests and routes them to the appropriate partition server using the mapping managed by the partition manager.

Although we have only coarsely described the architecture of AS, it's a great showcase of the scalability patterns applied to a concrete system. As an interesting historical note, AS was built from the ground up to be strongly consistent, while AWS S3 started of-

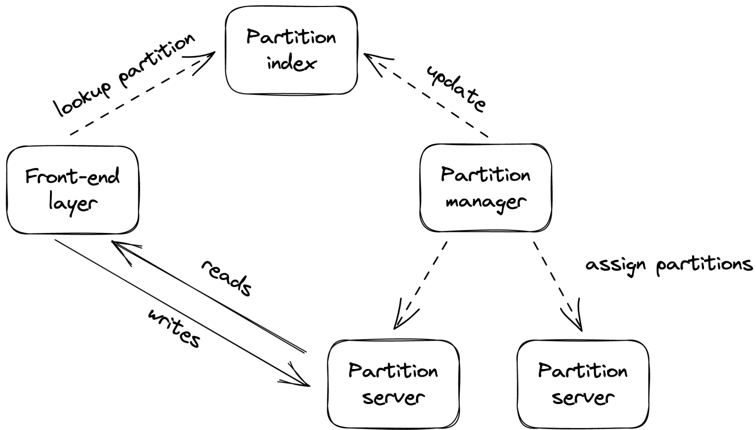

Figure 17.3: The partition manager range-partitions files across partition servers and rebalances the partitions when necessary.

fering the same guarantee in 2021[5].

[5] "Diving Deep on S3 Consistency," https://www.allthingsdistributed.com/2021/04/s3-strong-consistency.html

Chapter 18

Network load balancing

By offloading requests to the file store and the CDN, *Cruder* is able to serve significantly more requests than before. But the free lunch is only going to last so long. Because there is a single application server, it will inevitably fall over if the number of requests directed at it keeps increasing. To avoid that, we can create multiple application servers, each running on a different machine, and have a *load balancer* distribute requests to them. The thinking is that if one server has a certain capacity, then, in theory, two servers should have twice that capacity. This is an example of the more general scalability pattern we referred to as scaling out or scaling horizontally.

The reason we can scale *Cruder* horizontally is that we have pushed the state to dedicated services (the database and the managed file store). Scaling out a stateless application doesn't require much effort, *assuming* its dependencies can scale accordingly as well. As we will discuss in the next chapter, scaling out a stateful service, like a data store, is a lot more challenging since it needs to replicate state and thus requires some form of coordination, which adds complexity and can also become a bottleneck. As a general rule of thumb, we should try to keep our applications stateless by pushing state to third-party services designed by teams with years of

experience building such services.

Distributing requests across a pool of servers has many benefits. Because clients are decoupled from servers and don't need to know their individual addresses, the number of servers behind the load balancer can increase or decrease transparently. And since multiple redundant servers can interchangeably be used to handle requests, a load balancer can detect faulty ones and take them out of the pool, increasing the availability of the overall application.

As you might recall from chapter 1, the availability of a system is the percentage of time it's capable of servicing requests and doing useful work. Another way of thinking about it is that it's the probability that a request will succeed.

The reason why a load balancer increases the theoretical availability is that in order for the application to be considered unavailable, all the servers need to be down. With N servers, the probability that they are all unavailable is the product of the servers' failure rates[1]. By subtracting this product from 1, we can determine the theoretical availability of the application.

For example, if we have two servers behind a load balancer and each has an availability of 99%, then the application has a theoretical availability of 99.99%:

$$1 - (0.01 \cdot 0.01) = 0.9999$$

Intuitively, the nines of independent servers sum up.[2] Thus, in the previous example, we have two independent servers with two nines each, for a total of four nines of availability. Of course, this number is only theoretical because, in practice, the load balancer doesn't remove faulty servers from the pool immediately. The formula also naively assumes that the failure rates are independent, which might not be the case. Case in point: when a faulty server is

[1] "AWS Well-Architected Framework, Availability," https://docs.aws.amazon.com/wellarchitected/latest/reliability-pillar/availability.html

[2] Another way to think about it is that by increasing the number of servers linearly, we increase the availability exponentially.

removed from the load balancer's pool, the remaining ones might not be able to sustain the increase in load and degrade.

In the following sections, we will take a closer look at some of the core features offered by a load balancer.

Load balancing

The algorithms used for routing requests can vary from round-robin to consistent hashing to ones that take into account the servers' load.

As a fascinating side note, balancing by load is a lot more challenging than it seems in a distributed context. For example, the load balancer could periodically sample a dedicated *load endpoint* exposed by each server that returns a measure of how busy the server is (e.g., CPU usage). And since constantly querying servers can be costly, the load balancer can cache the responses for some time.

Using cached or otherwise delayed metrics to distribute requests to servers can result in surprising behaviors. For example, if a server that just joined the pool reports a load of 0, the load balancer will hammer it until the next time its load is sampled. When that happens, the server will report that it's overloaded, and the load balancer will stop sending more requests to it. This causes the server to alternate between being very busy and not being busy at all.

As it turns out, randomly distributing requests to servers without accounting for their load achieves a better load distribution. Does that mean that load balancing using delayed load metrics is not possible? There is a way, but it requires combining load metrics with the power of randomness. The idea is to randomly pick two servers from the pool and route the request to the least-loaded one of the two. This approach works remarkably well in practice[3].

Service discovery

[3] "The power of two random choices," https://brooker.co.za/blog/2012/01/17/two-random.html

Service discovery is the mechanism the load balancer uses to discover the pool of servers it can route requests to. A naive way to implement it is to use a static configuration file that lists the IP addresses of all the servers, which is painful to manage and keep up to date.

A more flexible solution is to have a fault-tolerant coordination service, like, e.g., etcd or Zookeeper, manage the list of servers. When a new server comes online, it registers itself to the coordination service with a TTL. When the server unregisters itself, or the TTL expires because it hasn't renewed its registration, the server is removed from the pool.

Adding and removing servers dynamically from the load balancer's pool is a key functionality cloud providers use to implement autoscaling[4], i.e., the ability to spin up and tear down servers based on load.

Health checks

A load balancer uses health checks to detect when a server can no longer serve requests and needs to be temporarily removed from the pool. There are fundamentally two categories of health checks: passive and active.

A *passive health check* is performed by the load balancer as it routes incoming requests to the servers downstream. If a server isn't reachable, the request times out, or the server returns a non-retriable status code (e.g., `503`), the load balancer can decide to take that server out of the pool.

Conversely, an *active health check* requires support from the downstream servers, which need to expose a dedicated *health endpoint* that the load balancer can query periodically to infer the server's health. The endpoint returns *200 (OK)* if the server can serve requests or a 5xx status code if it's overloaded and doesn't have more capacity to serve requests. If a request to the endpoint times out, it also counts as an error.

[4] "Autoscaling," https://docs.microsoft.com/en-us/azure/architecture/best-practices/auto-scaling

The endpoint's handler could be as simple as always returning *200 OK*, since most requests will time out when the server is degraded. Alternatively, the handler can try to infer whether the server is degraded by comparing local metrics, like CPU usage, available memory, or the number of concurrent requests being served, with configurable thresholds.

But here be dragons[5]: if a threshold is misconfigured or the health check has a bug, all the servers behind the load balancer may fail the health check. In that case, the load balancer could naively empty the pool, taking the application down. However, in practice, if the load balancer is "smart enough," it should detect that a large fraction of the servers are unhealthy and consider the health checks to be unreliable. So rather than removing servers from the pool, it should ignore the health checks altogether so that new requests can be sent to any server.

Thanks to health checks, the application behind the load balancer can be updated to a new version without any downtime. During the update, a rolling number of servers report themselves as unavailable so that the load balancer stops sending requests to them. This allows in-flight requests to complete (drain) before the servers are restarted with the new version. More generally, we can use this mechanism to restart a server without causing harm.

For example, suppose a stateless application has a rare memory leak that causes a server's available memory to decrease slowly over time. When the server has very little physical memory available, it will swap memory pages to disk aggressively. This constant swapping is expensive and degrades the performance of the server dramatically. Eventually, the leak will affect the majority of servers and cause the application to degrade.

In this case, we could force a severely degraded server to restart. That way, we don't have to develop complex recovery logic when a server gets into a rare and unexpected degraded mode. Moreover, restarting the server allows the system to self-heal, giving its

[5] "Implementing health checks," https://aws.amazon.com/builders-library/implementing-health-checks/

operators time to identify the root cause.

To implement this behavior, a server could have a separate background thread — a *watchdog* — that wakes up periodically and monitors the server's health. For example, the watchdog could monitor the available physical memory left. When a monitored metric breaches a specific threshold for some time, the watchdog considers the server degraded and deliberately crashes or restarts it.

Of course, the watchdog's implementation needs to be well-tested and monitored since a bug could cause servers to restart continuously. That said, I find it uncanny how this simple pattern can make an application a lot more robust to gray failures.

18.1 DNS load balancing

Now that we are familiar with the job description of a load balancer, let's take a closer look at how it can be implemented. While you won't have to build your own load balancer given the abundance of off-the-shelf solutions available, it's important to have a basic knowledge of how a load balancer works. Because every request needs to go through it, it contributes to your applications' performance and availability.

A simple way to implement a load balancer is with DNS. For example, suppose we have a couple of servers that we would like to load-balance requests over. If these servers have public IP addresses, we can add those to the application's DNS record and have the clients pick one[6] when resolving the DNS address, as shown in Figure 18.1.

Although this approach works, it's not resilient to failures. If one of the two servers goes down, the DNS server will happily continue to serve its IP address, unaware that it's no longer available. Even if we were to automatically reconfigure the DNS record when a failure happens and take out the problematic IP, the change needs

[6] "Round-robin DNS," https://en.wikipedia.org/wiki/Round-robin_DNS

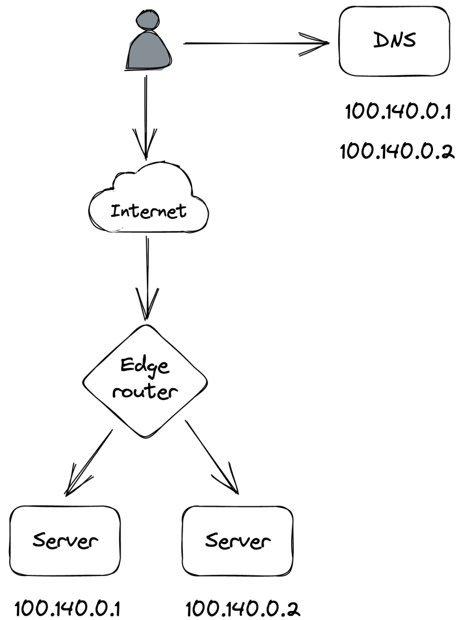

Figure 18.1: DNS load balancing

time to propagate to the clients, since DNS entries are cached, as discussed in chapter 4.

The one use case where DNS is used in practice to load-balance is for distributing traffic to different data centers located in different regions (*global DNS load balancing*). We have already encountered a use for this when discussing CDNs.

18.2 Transport layer load balancing

A more flexible load-balancing solution can be implemented with a load balancer that operates at the TCP level of the network stack (aka L4 load balancer[7]) through which all the traffic between clients and servers flows.

A network load balancer has one or more physical *network interface*

[7] layer 4 is the transport layer in the OSI model

cards mapped to one or more *virtual IP* (VIP) addresses. A VIP, in turn, is associated with a pool of servers. The load balancer acts as an intermediary between clients and servers — clients only see the VIP exposed by the load balancer and have no visibility of the individual servers associated with it.

When a client creates a new TCP connection with a load balancer's VIP, the load balancer picks a server from the pool and henceforth shuffles the packets back and forth for that connection between the client and the server. And because all the traffic goes through the load balancer, it can detect servers that are unavailable (e.g., with a passive health check) and automatically take them out of the pool, improving the system's reliability.

A connection is identified by a tuple (source IP/port, destination IP/port). Typically, some form of hashing is used to assign a connection tuple to a server that minimizes the disruption caused by a server being added or removed from the pool, like consistent hashing[8].

To forward packets downstream, the load balancer translates[9] each packet's source address to the load balancer's address and its destination address to the server's address. Similarly, when the load balancer receives a packet from the server, it translates its source address to the load balancer's address and its destination address to the client's address (see Figure 18.2).

As the data going out of the servers usually has a greater volume than the data coming in, there is a way for servers to bypass the load balancer and respond directly to the clients using a mechanism called direct server return[10], which can significantly reduce the load on the load balancer.

A network load balancer can be built using commodity machines

[8] "SREcon19 Americas - Keeping the Balance: Internet-Scale Loadbalancing Demystified," https://www.youtube.com/watch?v=woaGu3kJ-xk

[9] "Network address translation," https://en.wikipedia.org/wiki/Network_address_translation

[10] "Introduction to modern network load balancing and proxying," https://blog.envoyproxy.io/introduction-to-modern-network-load-balancing-and-proxying-a57f6ff80236

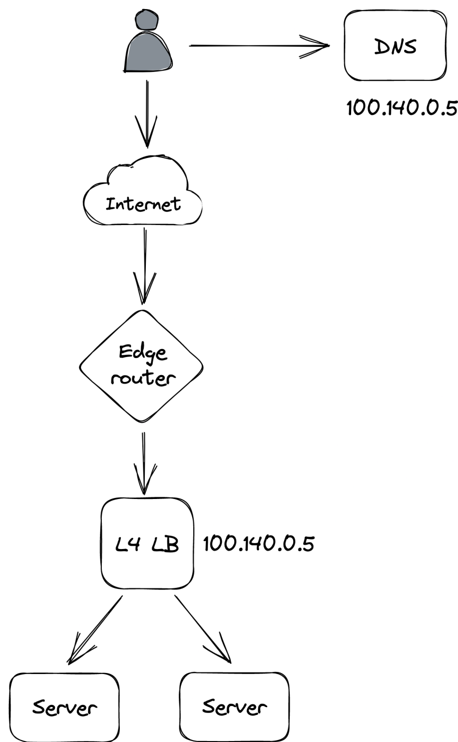

Figure 18.2: Transport layer load balancing

and scaled out using a combination of *Anycast*[11] and *ECMP*[12]. Load balancer instances announce themselves to the data center's edge routers with the same Anycast VIP and identical BGP weight. Using an Anycast IP is a neat trick that allows multiple machines to share the same IP address and have routers send traffic to the one with the lowest BGP weight. If all the instances have the same identical BGP weight, routers use equal-cost multi-path routing (consistent hashing) to ensure that the packets of a specific connection are generally routed to the same load balancer instance.

Since *Cruder* is hosted in the cloud, we can leverage one of the

[11] "Anycast," https://en.wikipedia.org/wiki/Anycast

[12] "Equal-cost multi-path routing," https://en.wikipedia.org/wiki/Equal-cost_multi-path_routing

many managed solutions for network load balancing, such as AWS Network Load Balancer[13] or Azure Load Balancer[14].

Although load balancing connections at the TCP level is very fast, the drawback is that the load balancer is just shuffling bytes around without knowing what they actually mean. Therefore, L4 load balancers generally don't support features that require higher-level network protocols, like terminating TLS connections. A load balancer that operates at a higher level of the network stack is required to support these advanced use cases.

18.3 Application layer load balancing

An application layer load balancer (aka L7 load balancer[15]) is an HTTP reverse proxy that distributes requests over a pool of servers. The load balancer receives an HTTP request from a client, inspects it, and sends it to a backend server.

There are two different TCP connections at play here, one between the client and the L7 load balancer and another between the L7 load balancer and the server. Because a L7 load balancer operates at the HTTP level, it can de-multiplex individual HTTP requests sharing the same TCP connection. This is even more important with HTTP 2, where multiple concurrent streams are multiplexed on the same TCP connection, and some connections can be a lot more expensive to handle than others.

The load balancer can do smart things with application traffic, like rate-limit requests based on HTTP headers, terminate TLS connections, or force HTTP requests belonging to the same *logical session* to be routed to the same backend server. For example, the load balancer could use a cookie to identify which logical session a request belongs to and map it to a server using consistent hashing. That allows servers to cache session data in memory and avoid fetching

[13]"Network Load Balancer," https://aws.amazon.com/elasticloadbalancing/network-load-balancer/

[14]"Azure Load Balancer," https://azure.microsoft.com/en-us/services/load-balancer/

[15]layer 7 is the application layer in the OSI model

it from the data store for each request. The caveat is that sticky sessions can create hotspots, since some sessions can be much more expensive to handle than others.

A L7 load balancer can be used as the backend of a L4 load balancer that load-balances requests received from the internet. Although L7 load balancers have more capabilities than L4 load balancers, they also have lower throughput, making L4 load balancers better suited to protect against certain DDoS attacks, like SYN floods[16].

A drawback of using a dedicated load balancer is that all the traffic directed to an application needs to go through it. So if the load balancer goes down, the application behind it does too. However, if the clients are internal to the organization, load balancing can be delegated to them using the *sidecar pattern*. The idea is to proxy all a client's network traffic through a process co-located on the same machine (the sidecar proxy). The sidecar process acts as a L7 load balancer, load-balancing requests to the right servers. And, since it's a reverse proxy, it can also implement various other functions, such as rate-limiting, authentication, and monitoring.

This approach[17] (aka "service mesh") has been gaining popularity with the rise of microservices in organizations with hundreds of services communicating with each other. As of this writing, popular sidecar proxy load balancers are NGINX, HAProxy, and Envoy. The main advantage of this approach is that it delegates load-balancing to the clients, removing the need for a dedicated load balancer that needs to be scaled out and maintained. The drawback is that it makes the system a lot more complex since now we need a control plane to manage all the sidecars[18].

[16] A SYN flood is a form of denial-of-service attack in which an attacker rapidly initiates a TCP connection to a server without finalizing the connection.

[17] "Service mesh data plane vs. control plane," https://blog.envoyproxy.io/service-mesh-data-plane-vs-control-plane-2774e720f7fc

[18] "Service Mesh Wars, Goodbye Istio," https://blog.polymatic.systems/service-mesh-wars-goodbye-istio-b047d9e533c7

Chapter 19

Data storage

Because *Cruder* is stateless, we were able to scale it out by running multiple application servers behind a load balancer. But as the application handles more load, the number of requests to the relational database increases as well. And since the database is hosted on a single machine, it's only a matter of time until it reaches its capacity and it starts to degrade.

19.1 Replication

We can increase the read capacity of the database by creating replicas. The most common way of doing that is with a leader-follower topology (see Figure 19.1). In this model, clients send writes (updates, inserts, and deletes) exclusively to the leader, which persists the changes to its write-ahead log. Then, the followers, or replicas, connect to the leader and stream log entries from it, committing them locally. Since log entries have a sequence number, followers can disconnect and reconnect at any time and start from where they left off by communicating to the leader the last sequence number they processed.

By creating read-only followers and putting them behind a load balancer, we can increase the read capacity of the database. Repli-

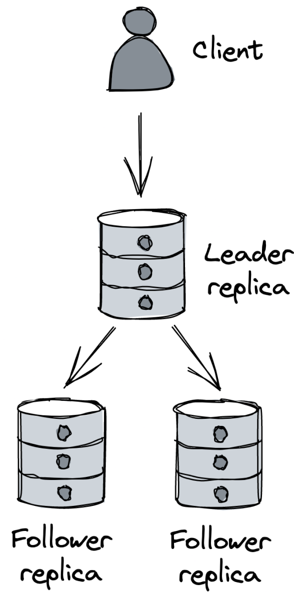

Figure 19.1: Single leader replication

cation also increases the availability of the database. For example, the load balancer can automatically take a faulty replica out of the pool when it detects that it's no longer healthy or available. And when the leader fails, a replica can be reconfigured to take its place. Additionally, individual followers can be used to isolate specific workloads, like expensive analytics queries that are run periodically, so that they don't impact the leader and other replicas.

The replication between the leader and the followers can happen either fully synchronously, fully asynchronously, or as a combination of the two.

If the replication is *fully asynchronous*, when the leader receives a write, it broadcasts it to the followers and immediately sends a response back to the client without waiting for the followers to acknowledge it. Although this approach minimizes the response time for the client, it's not fault-tolerant. For example, the leader could crash right after acknowledging a write and before broad-

casting it to the followers, resulting in data loss.

In contrast, if replication is *fully synchronous*, the leader waits for the write to be acknowledged by the followers before returning a response to the client. This comes with a performance cost since a single slow replica increases the response time of every request. And if any replica is unreachable, the data store becomes unavailable. This approach is not scalable; the more followers there are, the more likely it is that at least one of them is slow or unavailable.

In practice, relational databases often support a combination of synchronous and asynchronous replication. For example, in PostgreSQL, individual followers can be configured to receive updates synchronously[1], rather than asynchronously, which is the default. So, for example, we could have a single synchronous follower whose purpose is to act as an up-to-date backup of the leader. That way, if the leader fails, we can fail over to the synchronous follower without incurring any data loss.

Conceptually, the failover mechanism needs to: detect when the leader has failed, promote the synchronous follower to be the new leader and reconfigure the other replicas to follow it, and ensure client requests are sent to the new leader. Managed solutions like AWS RDS or Azure SQL Database, support read replicas[2] and automated failover[3] out of the box, among other features such as automated patching and backups.

One caveat of replication is that it only helps to scale out reads, not writes. The other issue is that the entire database needs to fit on a single machine. Although we can work around that by moving some tables from the main database to others running on different nodes, we would only be delaying the inevitable. As you should know by now, we can overcome these limitations with partitioning.

---

[1] "PostgreSQL Server Configuration, Replication," https://www.postgresql.org/docs/14/runtime-config-replication.html

[2] "Amazon RDS Read Replicas," https://aws.amazon.com/rds/features/read-replicas/

[3] "Multi-AZ deployments for high availability," https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/Concepts.MultiAZ.html

19.2 Partitioning

Partitioning allows us to scale out a database for both reads and writes. Even though traditional (centralized) relational databases generally don't support it out of the box, we can implement it at the application layer in principle. However, implementing partitioning at the application layer is challenging and adds a lot of complexity to the system. For starters, we need to decide how to partition the data among the database instances and rebalance it when a partition becomes too hot or too big. Once the data is partitioned, queries that span multiple partitions need to be split into sub-queries and their responses have to be combined (think of aggregations or joins). Also, to support atomic transactions across partitions, we need to implement a distributed transaction protocol, like 2PC. Add to all that the requirement to combine partitioning with replication, and you can see how partitioning at the application layer becomes daunting.

Taking a step back, the fundamental problem with traditional relational databases is that they have been designed under the assumption they fit on a single beefy machine. Because of that, they support a number of features that are hard to scale, like ACID transactions and joins. Relational databases were designed in an era where disk space was costly, and normalizing the data to reduce the footprint on disk was a priority, even if it came with a significant cost to unnormalize the data at query time with joins.[4]

Times have changed, and storage is cheap nowadays, while CPU time isn't. This is why, in the early 2000s, large tech companies began to build bespoke solutions for storing data designed from the ground up with high availability and scalability in mind.

[4]That said, reducing storage costs isn't the only benefit of normalization: it also helps maintain data integrity. If a piece of data is duplicated in multiple places, then to update it, we have to make sure it gets updated everywhere. In contrast, if the data is normalized, we only need to update it in one place.

19.3 NoSQL

These early solutions didn't support SQL and more generally only implemented a fraction of the features offered by traditional relational data stores. White papers such as Bigtable[5] and Dynamo[6] revolutionized the industry and started a push towards scalable storage layers, resulting in a plethora of open source solutions inspired by them, like HBase and Cassandra.

Since the first generation of these data stores didn't support SQL, they were referred to as *NoSQL*. Nowadays, the designation is misleading as NoSQL stores have evolved to support features, like dialects of SQL, that used to be available only in relational data stores.

While relational databases support stronger consistency models such as strict serializability, NoSQL stores embrace relaxed consistency models such as eventual and causal consistency to support high availability.

Additionally, NoSQL stores generally don't provide joins and rely on the data, often represented as key-value pairs or documents (e.g., JSON), to be unnormalized. A pure key-value store maps an opaque sequence of bytes (key) to an opaque sequence of bytes (value). A document store maps a key to a (possibly hierarchical) document without a strictly enforced schema. The main difference from a key-value store is that documents are interpreted and indexed and therefore can be queried based on their internal structure.

Finally, since NoSQL stores natively support partitioning for scalability purposes, they have limited support for transactions. For example, Azure Cosmos DB currently only supports transactions scoped to individual partitions. On the other hand, since the data is stored in unnormalized form, there is less need for transactions or joins in the first place.

[5] "Bigtable: A Distributed Storage System for Structured Data," https://static.googleusercontent.com/media/research.google.com/en//archive/bigtable-osdi06.pdf

[6] we talked about Dynamo in section 11.3

Although the data models used by NoSQL stores are generally not relational, we can still use them to model relational data. But if we take a NoSQL store and try to use it as a relational database, we will end up with the worst of both worlds. If used correctly, NoSQL can handle many of the use cases that a traditional relational database can[7], while being essentially scalable from day 1.[8]

The main requirement for using a NoSQL data store efficiently is to know the access patterns upfront and model the data accordingly; let's see why that is so important. Take Amazon DynamoDB[9], for example; its main abstraction is a table that contains items. Each item can have different attributes, but it must have a primary key that uniquely identifies an item.

The primary key can consist of either a single attribute, the *partition key*, or of two attributes, the *partition key* and the *sort key*. As you might suspect, the partition key dictates how the data is partitioned and distributed across nodes, while the sort key defines how the data is sorted within a partition, which allows for efficient range queries.

DynamoDB creates three replicas for each partition and uses state machine replication to keep them in sync[10]. Writes are routed to the leader, and an acknowledgment is sent to the client when two out of three replicas have received the write. Reads can be either eventually consistent (pick any replica) or strongly consistent (query the leader). Confusingly enough, the architecture of DynamoDB is very different from the one presented in the Dynamo paper, which we discussed in chapter 11.3.

At a high level, DynamoDB's API supports:

- CRUD operations on single items,

[7] "AWS re:Invent 2018: Amazon DynamoDB Deep Dive: Advanced Design Patterns for DynamoDB (DAT401)," https://www.youtube.com/watch?v=HaEPXoXVf2k

[8] On the other hand, while we certainly can find ways to scale a relational database, what works on day 1 might not work on day 10 or 100.

[9] "Amazon DynamoDB," https://aws.amazon.com/dynamodb/

[10] "AWS re:Invent 2018: Amazon DynamoDB Under the Hood: How We Built a Hyper-Scale Database (DAT321)," https://www.youtube.com/watch?v=yvBR71D0nAQ

- querying multiple items that have the same partition key (optionally specifying conditions on the sort key),
- and scanning the entire table.

There are no join operations, by design, since they don't scale well. But that doesn't mean we should implement joins in the application. Instead, as we will see shortly, we should model our data so that joins aren't needed in the first place.

The partition and sort key attributes are used to model the table's access patterns. For example, suppose the most common access pattern is retrieving the list of orders for a specific customer sorted by date. In that case, it would make sense for the table to have the customer ID as the partition key, and the order creation date as the sort key:

Partition Key	Sort Key	Attribute	Attribute
jonsnow	2021-07-13	OrderID: 1452	Status: Shipped
aryastark	2021-07-20	OrderID: 5252	Status: Placed
branstark	2021-07-22	OrderID: 5260	Status: Placed

Now suppose that we also want the full name of the customer in the list of orders. While, in a relational database, a table contains only entities of a certain type (e.g., customer), in NoSQL, a table can contain entities of multiple types. Thus, we could store both customers and orders within the same table:

Partition Key	Sort Key	Attribute	Attribute
jonsnow	2021-07-13	OrderID: 1452	Status: Shipped
jonsnow	jonsnow	FullName: Jon Snow	Address: …
aryastark	2021-07-20	OrderID: 5252	Status: Placed
aryastark	aryastark	FullName: Arya Stark	Address: …

Because a customer and its orders have the same partition key, we can now issue a single query that retrieves all entities for the desired customer.

See what we just did? We have structured the table based on the access patterns so that queries won't require any joins. Now think for a moment about how you would model the same data in normalized form in a relational database. You would probably have one table for orders and another for customers. And, to perform the same query, a join would be required at query time, which would be slower and harder to scale.

The previous example is very simple, and DynamoDB supports secondary indexes to model more complex access patterns — local secondary indexes allow for alternate sort keys in the same table, while global secondary indexes allow for different partition and sort keys, with the caveat that index updates are asynchronous and eventually consistent.

It's a common misconception that NoSQL data stores are more flexible than relational databases because they can seamlessly scale without modeling the data upfront. Nothing is further from the truth — NoSQL requires a lot more attention to how the data is modeled. Because NoSQL stores are tightly coupled to the access patterns, they are a lot less flexible than relational databases.

If there is one concept you should take away from this chapter, it's this: using a NoSQL data store requires identifying the access patterns upfront to model the data accordingly. If you want to learn how to do that, I recommend reading "The DynamoDB Book"[11], even if you plan to use a different NoSQL store.

As scalable data stores keep evolving, the latest trend is to combine the scalability of NoSQL with the ACID guarantees of relational databases. These new data stores are also referred to as NewSQL[12]. While NoSQL data stores prioritize availability over consistency in the face of network partitions, NewSQL stores prefer consistency. The argument behind NewSQL stores is that, with the right design, the reduction in availability caused by enforcing strong con-

[11] "The DynamoDB Book," https://www.dynamodbbook.com/

[12] "Andy Pavlo — The official ten-year retrospective of NewSQL databases," https://www.youtube.com/watch?v=LwkS82zs65g

sistency is hardly noticeable[13] for many applications. That, combined with the fact that perfect 100% availability is not possible anyway (availability is defined in 9s), has spurred the drive to build storage systems that can scale but favor consistency over availability in the presence of network partitions. CockroachDB[14] and Spanner[15] are well-known examples of NewSQL data stores.

[13] "NewSQL database systems are failing to guarantee consistency, and I blame Spanner," https://dbmsmusings.blogspot.com/2018/09/newsql-database-systems-are-failing-to.html

[14] "CockroachDB," https://github.com/cockroachdb/cockroach

[15] we talked about Spanner in section 12.4

Chapter 20

Caching

Suppose a significant fraction of requests that *Cruder* sends to its data store consists of a small pool of frequently accessed entries. In that case, we can improve the application's performance and reduce the load on the data store by introducing a cache. A *cache* is a high-speed storage layer that temporarily buffers responses from an *origin*, like a data store, so that future requests can be served directly from it. It only provides best-effort guarantees, since its state is disposable and can be rebuilt from the origin. We have already seen some applications of caching when discussing the DNS protocol or CDNs.

For a cache to be cost-effective, the proportion of requests that can be served directly from it (hit ratio) should be high. The *hit ratio* depends on several factors, such as the universe of cachable objects (the fewer, the better), the likelihood of accessing the same objects repeatedly (the higher, the better), and the size of the cache (the larger, the better).

As a general rule of thumb, the higher up in the call stack caching is used, the more resources can be saved downstream. This is why the first use case for caching we discussed was client-side HTTP caching. However, it's worth pointing out that caching is an optimization, and you don't have a scalable architecture if the origin,

e.g., the data store in our case, can't withstand the load without the cache fronting it. If the access pattern suddenly changes, leading to cache misses, or the cache becomes unavailable, you don't want your application to fall over (but it's okay for it to become slower).

20.1 Policies

When a cache miss occurs, the missing object has to be requested from the origin, which can happen in two ways:

- After getting an "object-not-found" error from the cache, the application requests the object from the origin and updates the cache. In this case, the cache is referred to as a *side cache*, and it's typically treated as a key-value store by the application.
- Alternatively, the cache is *inline*, and it communicates directly with the origin, requesting the missing object on behalf of the application. In this case, the application only ever accesses the cache. We have already seen an example of an inline cache when discussing HTTP caching.

Because a cache has a limited capacity, one or more entries need to be evicted to make room for new ones when its capacity is reached. Which entry to remove depends on the eviction policy used by the cache and the objects' access pattern. For example, one commonly used policy is to evict the *least recently used* (LRU) entry.

A cache can also have an *expiration policy* that dictates when an object should be evicted, e.g., a TTL. When an object has been in the cache for longer than its TTL, it expires and can safely be evicted. The longer the expiration time, the higher the hit ratio, but also the higher the likelihood of serving stale and inconsistent data.

The expiration doesn't need to occur immediately, and it can be deferred to the next time the entry is requested. In fact, that might be preferable — if the origin (e.g., a data store) is temporarily unavailable, it's more resilient to return an object with an expired TTL to the application rather than an error.

An expiry policy based on TTL is a workaround for *cache invalidation*, which is very hard to implement in practice[1]. For example, if you were to cache the result of a database query, every time any of the data touched by that query changes (which could span thousands of records or more), the cached result would need to be invalidated somehow.

20.2 Local cache

The simplest way to implement a cache is to co-locate it with the client. For example, the client could use a simple in-memory hash table or an embeddable key-value store, like RocksDB[2], to cache responses (see Figure 20.1).

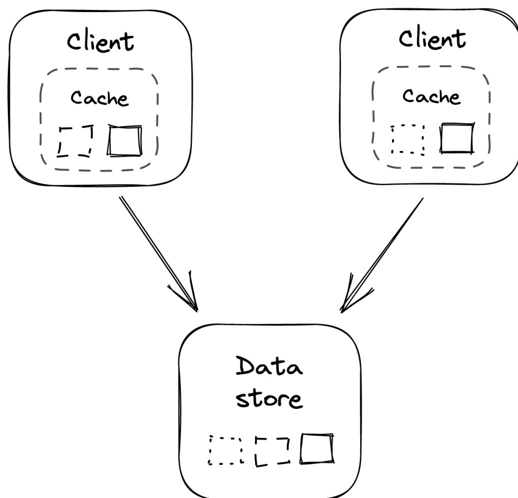

Figure 20.1: In-process cache

Because each client cache is independent of the others, the same objects are duplicated across caches, wasting resources. For example, if every client has a local cache of 1GB, then no matter how many clients there are, the total size of the cache is 1 GB. Also, con-

[1] "Cache coherence," https://en.wikipedia.org/wiki/Cache_coherence
[2] "RocksDB," http://rocksdb.org/

sistency issues will inevitably arise; for example, two clients might see different versions of the same object.

Additionally, as the number of clients grows, the number of requests to the origin increases. This issue is exacerbated when clients restart, or new ones come online, and their caches need to be populated from scratch. This can cause a "thundering herd" effect where the downstream origin is hit with a spike of requests. The same can also happen when a specific object that wasn't accessed before becomes popular all of a sudden.

Clients can reduce the impact of a thundering herd by *coalescing* requests for the same object. The idea is that, at any given time, there should be at most one outstanding request per client to fetch a specific object.

20.3 External cache

An external cache is a service dedicated to caching objects, typically in memory. Because it's shared across clients, it addresses some of the drawbacks of local caches at the expense of greater complexity and cost (see Figure 20.2). For example, Redis[3] or Memcached[4] are popular caching services, also available as managed services on AWS and Azure.

Unlike a local cache, an external cache can increase its throughput and size using replication and partitioning. For example, Redis[5] can automatically partition data across multiple nodes and replicate each partition using a leader-follower protocol.

Since the cache is shared among its clients, there is only a single version of each object at any given time (assuming the cache is not replicated), which reduces consistency issues. Also, the number of times an object is requested from the origin doesn't grow with the number of clients.

[3] "Redis," https://redis.io/
[4] "Memcached," https://memcached.org/
[5] "Redis cluster tutorial," https://redis.io/topics/cluster-tutorial

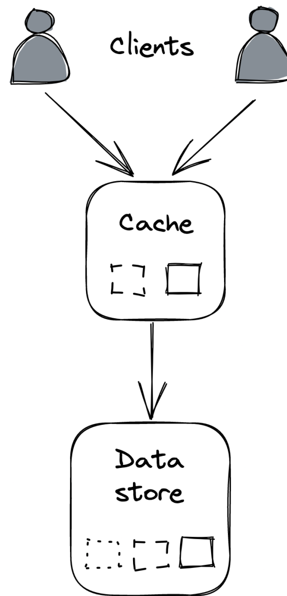

Figure 20.2: Out-of-process cache

Although an external cache decouples clients from the origin, the load merely shifts to the external cache. Therefore, the cache will eventually need to be scaled out if the load increases. When that happens, as little data as possible should be moved around (or dropped) to avoid the cache degrading or the hit ratio dropping significantly. Consistent hashing, or a similar partitioning technique, can help reduce the amount of data that needs to be shuffled when the cache is rebalanced.

An external cache also comes with a maintenance cost as it's yet another service that needs to be operated. Additionally, the latency to access it is higher than accessing a local cache because a network call is required.

If the external cache is down, how should the clients react? You would think it might be okay to bypass the cache and directly hit the origin temporarily. But the origin might not be prepared to withstand a sudden surge of traffic. Consequently, the external

cache becoming unavailable could cause a cascading failure, resulting in the origin becoming unavailable as well.

To avoid that, clients could use an in-process cache as a defense against the external cache becoming unavailable. That said, the origin also needs to be prepared to handle these sudden "attacks" by, e.g., shedding requests; we will discuss a few approaches for achieving that in the book's resiliency part. What's important to remember is that caching is an optimization, and the system needs to survive without it at the cost of being slower.

Chapter 21

Microservices

If *Cruder* is successful in the market, we can safely assume that we will continue to add more components to it to satisfy an ever-growing list of business requirements, as shown in Figure 21.1.

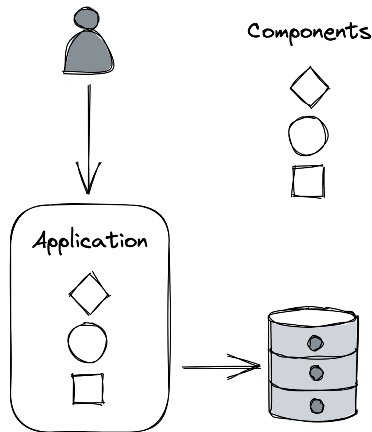

Figure 21.1: A monolithic application composed of multiple components

The components will likely become increasingly coupled in time, causing developers to step on each other's toes more frequently. Eventually, the codebase becomes complex enough that nobody

fully understands every part of it, and implementing new features or fixing bugs becomes a lot more time-consuming than it used to be.

Also, a change to a component might require the entire application to be rebuilt and deployed. And if the deployment of a new version introduces a bug, like a memory or socket leak, unrelated components might also be affected. Moreover, reverting a deployment affects the velocity of every developer, not just the one that introduced a bug.

One way to mitigate the growing pains of a *monolithic* application is to functionally decompose it into a set of independently deployable services that communicate via APIs, as shown in Figure 21.2. The APIs decouple the services from each other by creating boundaries that are hard to violate, unlike the ones between components running in the same process.

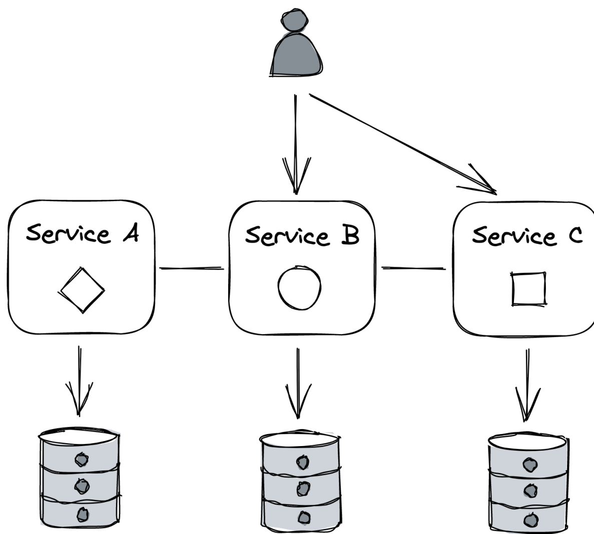

Figure 21.2: An application split into independently deployable services that communicate via APIs

Each service can be owned and operated by a small team. Smaller teams collaborate more effectively than larger ones, since the com-

munication overhead grows quadratically[1] with the team's size. And because each team controls its own codebase and dictates its own release schedule, less cross-team communication is required overall. Also, the surface area of a service is smaller than the whole application, making it more digestible to developers, especially new hires.

Each team is also free in principle to adopt the tech stack and hardware that fits their specific needs. After all, the consumers of the APIs don't care how the functionality is implemented. This makes it easy to experiment and evaluate new technologies without affecting other parts of the system. As a result, each service can have its own independent data model and data store(s) that best fit its use cases.

This architectural style is also referred to as the *microservice architecture*. The term *micro* is misleading, though — there doesn't have to be anything micro about services.[2] If a service doesn't do much, it only adds operational overhead and complexity. As a rule of thumb, APIs should have a small surface area and encapsulate a significant amount of functionality.[3]

21.1 Caveats

To recap, splitting an application into services adds a great deal of complexity to the overall system, which is only worth paying if it can be amortized across many development teams. Let's take a closer look at why that is.

Tech stack

While nothing forbids each microservice to use a different tech stack, doing so makes it more difficult for a developer to move

[1] "The Mythical Man-Month," https://en.wikipedia.org/wiki/The_Mythical_Man-Month

[2] A more appropriate name for the microservices architecture is service-oriented architecture, but unfortunately, that name comes with some baggage as well.

[3] This idea is described well in John Ousterhout's *A Philosophy of Software Design*, which I highly recommend.

from one team to another. And think of the sheer number of libraries — one for each language adopted — that need to be supported to provide common functionality that all services need, like logging.

It's only reasonable, then, to enforce a certain degree of standardization. One way to do that, while still allowing some degree of freedom, is to loosely encourage specific technologies by providing a great development experience for the teams that stick with the recommended portfolio of languages and technologies.

Communication

Remote calls are expensive and introduce non-determinism. Much of what is described in this book is about dealing with the complexity of distributed processes communicating over the network. That said, a monolith doesn't live in isolation either, since it serves external requests and likely depends on third-party APIs as well, so these issues need to be tackled there as well, albeit on a smaller scale.

Coupling

Microservices should be loosely coupled so that a change in one service doesn't require changing others. When that's not the case, you can end up with a dreaded distributed monolith, which has all the downsides of a monolith while being an order of magnitude more complex due to its distributed nature.

There are many causes of tight coupling, like fragile APIs that require clients to be updated whenever they change, shared libraries that have to be updated in lockstep across multiple services, or the use of static IP addresses to reference external services.

Resource provisioning

To support a large number of independent services, it should be simple to provision new machines, data stores, and other commodity resources — you don't want every team to come up with their own way of doing it. And, once these resources have been provisioned, they have to be configured. To pull this off efficiently, a

fair amount of automation is needed.

Testing

While testing individual microservices is not necessarily more challenging than testing a monolith, testing the integration of microservices is a lot harder. This is because very subtle and unexpected behaviors will emerge only when services interact with each other at scale in production.

Operations

Just like with resource provisioning, there should be a common way of continuously delivering and deploying new builds safely to production so that each team doesn't have to reinvent the wheel.

Additionally, debugging failures, performance degradations, and bugs is a lot more challenging with microservices, as you can't just load the whole application onto your local machine and step through it with a debugger. This is why having a good observability platform becomes crucial.

Eventual consistency

As a side effect of splitting an application into separate services, the data model no longer resides in a single data store. However, as we have learned in previous chapters, atomically updating data spread in different data stores, and guaranteeing strong consistency, is slow, expensive, and hard to get right. Hence, this type of architecture usually requires embracing eventual consistency.

So to summarize, it's generally best to start with a monolith and decompose it only when there is a good reason to do so[4]. As a bonus, you can still componentize the monolith, with the advantage that it's much easier to move the boundaries as the application grows. Once the monolith is well matured and growing pains start to arise, you can start to peel off one microservice at a time from it.

[4] "MicroservicePremium," https://martinfowler.com/bliki/MicroservicePremium.html

21.2 API gateway

After decomposing *Cruder* into a group of services, we need to rethink how the outside world communicates with the application. For example, a client might need to perform multiple requests to different services to fetch all the information it needs to complete a specific operation. This can be expensive on mobile devices, where every network request consumes precious battery life.

Moreover, clients need to be aware of implementation details, such as the DNS names of all the internal services. This makes it challenging to change the application's architecture as it requires changing the clients as well, which is hard to do if you don't control them. Once a public API is out there, you had better be prepared to maintain it for a very long time.

As is common in computer science, we can solve almost any problem by adding a layer of indirection. We can hide the internal APIs behind a public one that acts as a facade, or proxy, for the internal services (see Figure 21.3). The service that exposes this public API is called the *API gateway* (a reverse proxy).

21.2.1 Core responsibilities

Let's have a look at some of the most common responsibilities of an API gateway.

Routing

The most obvious function of an API gateway is routing inbound requests to internal services. One way to implement that is with the help of a routing map, which defines how the public API maps to the internal APIs. This mapping allows internal APIs to change without breaking external clients. For example, suppose there is a 1:1 mapping between a specific public endpoint and an internal one — if in the future the internal endpoint changes, the external clients can continue to use the public endpoint as if nothing had changed.

Composition

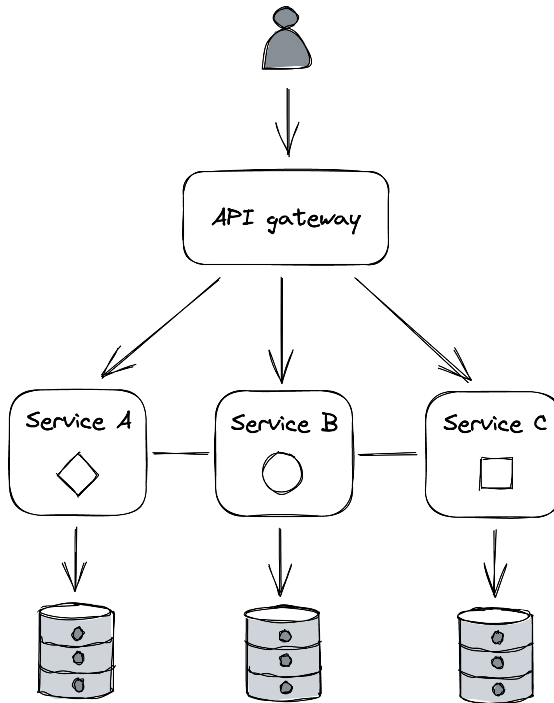

Figure 21.3: The API gateway hides the internal APIs from its clients.

The data of a monolithic application generally resides in a single data store, but in a distributed system, it's spread across multiple services, each using its own data store. As such, we might encounter use cases that require stitching data together from multiple sources. The API gateway can offer a higher-level API that queries multiple services and composes their responses. This relieves the client from knowing which services to query and reduces the number of requests it needs to perform to get the data it needs.

Composing APIs is not simple. The availability of the composed API decreases as the number of internal calls increases, since each has a non-zero probability of failure. Moreover, the data might be inconsistent, as updates might not have propagated to all services yet; in that case, the gateway will have to resolve this discrepancy

somehow.

Translation

The API gateway can translate from one IPC mechanism to another. For example, it can translate a RESTful HTTP request into an internal gRPC call.

It can also expose different APIs to different clients. For example, the API for a desktop application could potentially return more data than the one for a mobile application, as the screen estate is larger and more information can be presented at once. Also, network calls are more expensive for mobile clients, and requests generally need to be batched to reduce battery usage.

To meet these different and competing requirements, the gateway can provide different APIs tailored to different use cases and translate these to internal calls. Graph-based APIs are an increasingly popular solution for this. A *graph-based API* exposes a schema composed of types, fields, and relationships across types, which describes the data. Based on this schema, clients send queries declaring precisely what data they need, and the gateway's job is to figure out how to translate these queries into internal API calls.

This approach reduces the development time as there is no need to introduce different APIs for different use cases, and clients are free to specify what they need. There is still an API, though; it just happens that it's described with a graph schema, and the gateway allows to perform restricted queries on it. GraphQL[5] is the most popular technology in this space at the time of writing.

21.2.2 Cross-cutting concerns

As the API gateway is a reverse proxy, it can also implement cross-cutting functionality that otherwise would have to be part of each service. For example, it can cache frequently accessed resources or rate-limit requests to protect the internal services from being overwhelmed.

[5] "GraphQL," https://graphql.org/

Authentication and authorization are some of the most common and critical cross-cutting concerns. *Authentication* is the process of validating that a so-called *principal* — a human or an application — issuing a request is who it says it is. *Authorization* is the process of granting the authenticated principal permissions to perform specific operations, like creating, reading, updating, or deleting a particular resource. Typically, this is implemented by assigning one or more roles that grant specific permissions to a principal.

A common way for a monolithic application to implement authentication and authorization is with sessions. Because HTTP is a stateless protocol, the application needs a way to store data between HTTP requests to associate a request with any other request. When a client first sends a request to the application, the application creates a session object with an ID (e.g., a cryptographically-strong random number) and stores it in an in-memory cache or an external data store. The session ID is returned in the response through an HTTP cookie so that the client will include it in all future requests. That way, when the application receives a request with a session cookie, it can retrieve the corresponding session object.

So when a client sends its credentials to the application API's login endpoint, and the credential validation is successful, the principal's ID and roles are stored in the session object. The application can later retrieve this information and use it to decide whether to allow the principal to perform a request or not.

Translating this approach to a microservice architecture is not that straightforward. For example, it's not obvious which service should be responsible for authenticating and authorizing requests, as the handling of requests can span multiple services.

One approach is to have the API gateway authenticate external requests, since that's their point of entry. This allows centralizing the logic to support different authentication mechanisms into a single component, hiding the complexity from internal services. In contrast, *authorizing* requests is best left to individual services to avoid coupling the API gateway with domain logic.

When the API gateway has authenticated a request, it creates a *security token*. The gateway passes this token with the request to the internal services, which in turn pass it downstream to their dependencies. Now, when an internal service receives a request with a security token attached, it needs to have a way to validate it and obtain the principal's identity and roles. The validation differs depending on the type of token used, which can be *opaque* and not contain any information, or *transparent* and embed the principal's information within the token itself. The downside of an opaque token is that it requires calling an external auth service to validate it and retrieve the principal's information. Transparent tokens eliminate that call at the expense of making it harder to revoke compromised tokens.

The most popular standard for transparent tokens is the *JSON Web Token*[6] (JWT). A JWT is a JSON payload that contains an expiration date, the principal's identity and roles, and other metadata. In addition, the payload is signed with a certificate trusted by the internal services. Hence, no external calls are needed to validate the token.

Another common mechanism for authentication is the use of API keys. An *API key* is a custom key that allows the API gateway to identify the principal making a request and limit what they can do. This approach is popular for public APIs, like the ones of, e.g., Github or Twitter.

We have barely scratched the surface of the topic, and there are entire books[7] written on the subject that you can read to learn more about it.

21.2.3 Caveats

One of the drawbacks of using an API gateway is that it can become a development bottleneck. Since it's tightly coupled with the APIs of the internal services it's shielding, whenever an internal

[6] "Introduction to JSON Web Tokens," https://jwt.io/introduction

[7] "Microservices Security in Action," https://www.manning.com/books/microservices-security-in-action

API changes, the gateway needs to be modified as well. Another downside is that it's one more service that needs to be maintained. It also needs to scale to whatever the request rate is for all the services behind it.

That said, if an application has many services and APIs, the pros outweigh the cons, and it's generally a worthwhile investment. So how do you go about implementing a gateway? You can roll your own API gateway, using a reverse proxy as a starting point, like NGINX, or use a managed solution, like Azure API Management[8] or Amazon API Gateway[9].

[8] "Azure API Management," https://azure.microsoft.com/en-gb/services/api-management/

[9] "Amazon API Gateway," https://aws.amazon.com/api-gateway/

Chapter 22

Control planes and data planes

The API gateway is a single point of failure. If it goes down, then so does *Cruder*, which is why it needs to be highly available. And because every external request needs to go through it, it must also be scalable. This creates some interesting challenges with regard to external dependencies.

For example, suppose the gateway has a specific "configuration" or management endpoint to add, remove and configure API keys used to rate-limit requests. Unsurprisingly, the request volume for the configuration endpoint is a lot lower than the one for the main endpoint(s), and a lower scale would suffice to handle it. And while the gateway needs to prefer availability and performance over consistency for routing external requests to internal services, it should prefer consistency over availability for requests sent to the management endpoint.

Because of these different and competing requirements, we could split the API gateway into a *data plane* service that serves external requests directed towards our internal services and a *control plane* service that manages the gateway's metadata and configuration.

As it turns out, this split is a common pattern[1]. For example, in chain replication in section 10.4, the control plane holds the configuration of the chains. And in Azure Storage, which we discussed in chapter 17, the stream and partition managers are control planes that manage the allocation of streams and partitions to storage and partition servers, respectively.

More generally, a data plane includes any functionality on the critical path that needs to run for each client request. Therefore, it must be highly available, fast, and scale with the number of requests. In contrast, a control plane is not on the critical path and has less strict scaling requirements. Its main job is to help the data plane do its work by managing metadata or configuration and coordinating complex and infrequent operations. And since it generally needs to offer a consistent view of its state to the data plane, it favors consistency over availability.

An application can have multiple independent control and data planes. For example, a control plane might be in charge of scaling a service up or down based on load, while another manages its configuration.

But separating the control plane from the data plane introduces complexity. The data plane needs to be designed to withstand control plane failures for the separation to be robust. If the data plane stops serving requests when the control plane becomes unavailable, we say the former has a *hard dependency* on the latter. Intuitively, the entire system becomes unavailable if either the control plane or the data plane fails. More formally, when we have a chain of components that depend on each other, the theoretical availability of the system is the product of the availabilities of its components.

For example, if the data plane has a theoretical availability of 99.99%, but the control plane has an availability of 99%, then the overall system can only achieve a combined availability of 98.99%:

[1] "Control Planes vs Data Planes," https://brooker.co.za/blog/2019/03/17/control.html

$$0.9999 \cdot 0.99 = 0.9899$$

In other words, a system can at best be only as available as its least available hard dependency. We can try to make the control plane more reliable, but more importantly, we should ensure that the data plane can withstand control plane failures. If the control plane is temporarily unavailable, the data plane should continue to run with a stale configuration rather than stop. This concept is also referred to as *static stability*.

22.1 Scale imbalance

Generally, data planes and control planes tend to have very different scale requirements. This creates a risk as the data plane can overload[2] the control plane.

Suppose the control plane exposes an API that the data plane periodically queries to retrieve the latest configuration. Under normal circumstances, you would expect the requests to the control plane to be spread out in time more or less uniformly. But, in some cases, they can cluster within a short time interval. For example, if, for whatever reason, the processes that make up the data plane are restarted at the same time and must retrieve the configuration from the control plane, they could overload it.

Although the control plane can defend itself to some degree with the resiliency mechanisms described in chapter 28, eventually, it will start to degrade. If the control plane becomes unavailable because of overload or any other reason (like a network partition), it can take down the data plane with it.

Going back to the previous example, if part of the data plane is trying to start but can't reach the control plane because it's overloaded, it won't be able to come online. So how can we design around that?

[2] "Avoiding overload in distributed systems by putting the smaller service in control," https://aws.amazon.com/builders-library/avoiding-overload-in-distributed-systems-by-putting-the-smaller-service-in-control/

One way is to use a scalable file store, like Azure Storage or S3, as a buffer between the control plane and the data plane. The control plane periodically dumps its entire state to the file store regardless of whether it changed, while the data plane reads the state periodically from it (see Figure 22.1). Although this approach sounds naive and expensive, it tends to be reliable and robust in practice. And, depending on the size of the state, it might be cheap too.[3]

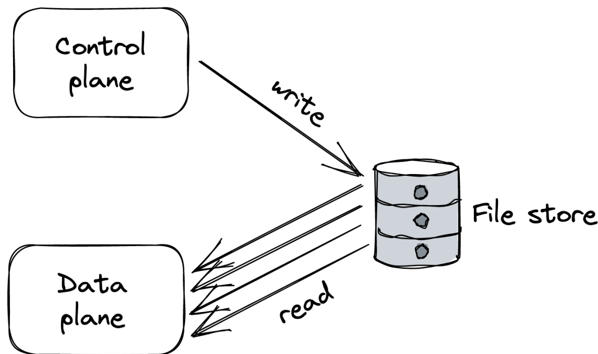

Figure 22.1: The intermediate data store protects the control plane by absorbing the load generated by the data plane.

Introducing an intermediate store as a buffer decouples the control plane from the data plane and protects the former from overload. It also enables the data plane to continue to operate (or start) if the control plane becomes unavailable. But this comes at the cost of higher latencies and weaker consistency guarantees, since the time it takes to propagate changes from the control plane to the data plane will necessarily increase.

To decrease the propagation latency, a different architecture is needed in which there is no intermediary. The idea is to have the data plane connect to the control plane, like in our original approach, but have the control plane push the configuration whenever it changes, rather than being at the mercy of periodic queries from the data plane. Because the control plane controls the pace, it will slow down rather than fall over when it can't keep

[3]This is another example of the CQRS pattern applied in practice.

up.[4]

To further reduce latencies and load, the control plane can version changes and push only updates/deltas to the data plane. Although this approach is more complex to implement, it significantly reduces the propagation time when the state is very large.

However, the control plane could still get hammered if many data plane instances start up around the same time (due to a massive scale-out or restart) and try to read the entire configuration from the control plane for the first time. To defend against this, we can reintroduce an intermediate data store that contains a recent snapshot of the control plane's state. This allows the data plane to read a snapshot from the store at startup and then request only a small delta from the control plane (see Figure 22.2).

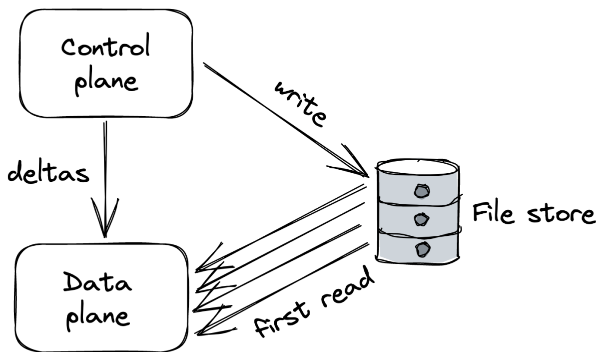

Figure 22.2: The intermediate data store absorbs the load of bulk reads, while the control plane pushes small deltas to the data plane whenever the state changes.

[4]That said, there is still the potential to overload the control plane if it needs to juggle too many connections.

22.2 Control theory

Control theory gives us another[5] way to think about control planes and data planes. In control theory, the goal is to create a controller that monitors a dynamic system, compares its state to the desired one, and applies a corrective action to drive the system closer to it while minimizing any instabilities on the way.

In our case, the data plane is the dynamic system we would like to drive to the desired state, while the controller is the control plane responsible for *monitoring* the data plane, *comparing* it to the desired state, and executing a corrective *action* if needed.

The control plane and the data plane are part of a feedback loop. And without all three ingredients (*monitor*, *compare*, and *action*), you don't have a closed loop, and the data plane can't reach the desired state[6]. The monitoring part is the most commonly missing ingredient to achieve a closed loop.

Take chain replication, for example. The control plane's job shouldn't be just to push the configuration of the chains to the data plane. It should also monitor whether the data plane has actually applied the configuration within a reasonable time. If it hasn't, it should perform some corrective action, which could be as naive as rebooting nodes with stale configurations or excluding them from being part of any chain.

A more mundane example of a control plane is a CI/CD pipeline for releasing a new version of a service without causing any disruption. One way to implement the pipeline is to deploy and release a new build blindly without monitoring the running service — the build might throw an exception at startup that prevents the service from starting, resulting in a catastrophic failure. Instead, the pipeline should release the new build incrementally while monitoring the service and stop the roll-out if there is clear evidence that

[5] "AWS re:Invent 2018: Close Loops & Opening Minds: How to Take Control of Systems, Big & Small ARC337," https://www.youtube.com/watch?v=O8xLxNje30M

[6] That said, having a closed loop doesn't guarantee that either, it's merely a prerequisite.

something is off, and potentially also roll it back automatically.

To sum up, when dealing with a control plane, ask yourself what's missing to close the loop. We have barely scratched the surface of the topic, and if you want to learn more about it, "Designing Distributed Control Systems"[7] is a great read.

[7] "Designing Distributed Control Systems," https://www.amazon.com/gp/product/1118694155

Chapter 23

Messaging

Suppose *Cruder* has an endpoint that allows users to upload a video and encode it in different formats and resolutions tailored to specific devices (TVs, mobile phones, tablets, etc.). When the API gateway receives a request from a client, it uploads the video to a file store, like S3, and sends a request to an encoding service to process the file.

Since the encoding can take minutes to complete, we would like the API gateway to send the request to the encoding service without waiting for the response. But the naive implementation (fire-and-forget) would cause the request to be lost if the encoding service instance handling it on the other side crashes or fails for any reason.

A more robust solution is to introduce a message channel between the API gateway and the encoding service. Messaging was first introduced in chapter 5 when discussing APIs. It's a form of indirect communication in which a producer writes a message to a channel — or message broker — that delivers the message to a consumer on the other end.

A message channel acts as a temporary buffer for the receiver. Unlike the direct request-response communication style we have been using so far, messaging is inherently asynchronous, as sending a

message doesn't require the receiving service to be online. The messages themselves have a well-defined format, consisting of a header and a body. The message header contains metadata, such as a unique message ID, while the body contains the actual content.

Typically, a message can either be a command, which specifies an operation to be invoked by the receiver, or an event, which signals the receiver that something of interest happened to the sender. A service can use inbound adapters to receive messages from channels and outbound adapters to send messages to channels, as shown in Figure 23.1.

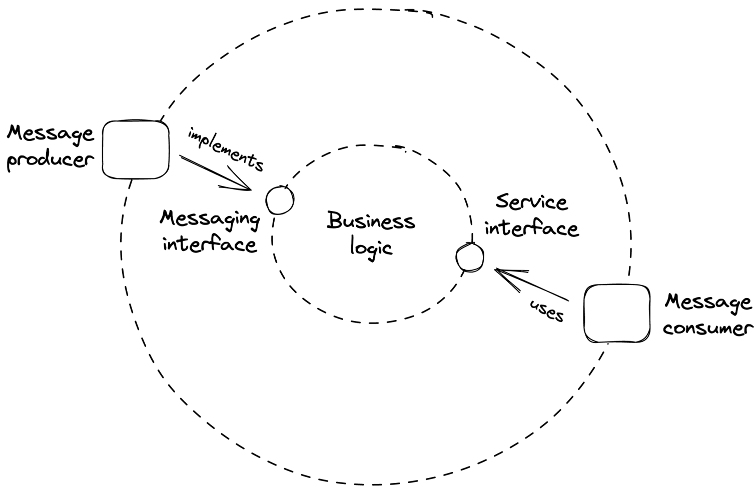

Figure 23.1: The message consumer (an inbound adapter) is part of the API surface of the service.

After the API gateway has uploaded the video to the file store, it writes a message to the channel with a link to the uploaded file and responds with *202 Accepted*, signaling to the client that the request has been accepted for processing but hasn't completed yet. Eventually, the encoding service will read the message from the channel and process it. Because the request is deleted from the channel only when it's successfully processed, the request will eventually be picked up again and retried if the encoding service fails to han-

dle it.

Decoupling the API gateway (producer) from the encoding service (consumer) with a channel provides many benefits. The producer can send requests to the consumer even if the consumer is temporarily unavailable. Also, requests can be load-balanced across a pool of consumer instances, making it easy to scale out the consuming side. And because the consumer can read from the channel at its own pace, the channel smooths out load spikes, preventing it from getting overloaded.

Another benefit is that messaging enables to process multiple messages within a single *batch* or *unit of work*. Most messaging brokers support this pattern by allowing clients to fetch up to N messages with a single read request. Although batching degrades the processing latency of individual messages, it dramatically improves the application's throughput. So when we can afford the extra latency, batching is a no brainer.

In general, a message channel allows any number of producer and consumer instances to write and read from it. But depending on how the channel delivers a message, it's classified as either point-to-point or publish-subscribe. In a *point-to-point* channel, the message is delivered to exactly one consumer instance. Instead, in a *publish-subscribe* channel, each consumer instance receives a copy of the message.

Unsurprisingly, introducing a message channel adds complexity. The message broker is yet another service that needs to be maintained and operated. And because there is an additional hop between the producer and consumer, the communication latency is necessarily going to be higher; more so if the channel has a large backlog of messages waiting to be processed. As always, it's all about tradeoffs.

Because messaging is a core pattern of distributed systems, we will take a closer look at it in this chapter, starting with the most common communication styles it enables.

One-way messaging

In this messaging style, the producer writes a message to a point-to-point channel with the expectation that a consumer will eventually read and process it (see Figure 23.2). This is the style we used in the example earlier.

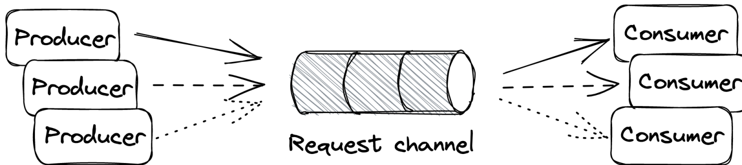

Figure 23.2: One-way messaging style

Request-response messaging

This messaging style is similar to the direct request-response style we are familiar with, albeit with the difference that the request and response messages flow through channels. The consumer has a point-to-point request channel from which it reads messages, while every producer has its dedicated response channel (see Figure 23.3).

When a producer writes a message to the request channel, it decorates it with a request ID and a reference to its response channel. Then, after a consumer has read and processed the message, it writes a reply to the producer's response channel, tagging it with the request's ID, which allows the producer to identify the request it belongs to.

Broadcast messaging

In this messaging style, the producer writes a message to a publish-subscribe channel to broadcast it to all consumer instances (see Figure 23.4). This style is generally used to notify a group of processes that a specific event has occurred. For example, we have already encountered this pattern when discussing the outbox pattern in section 13.1.

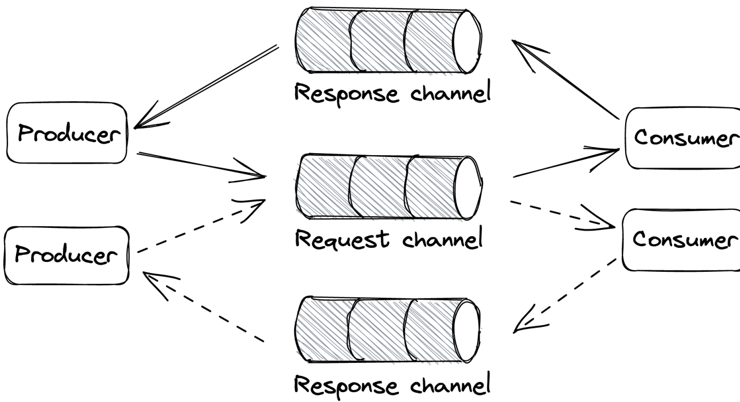

Figure 23.3: Request-response messaging style

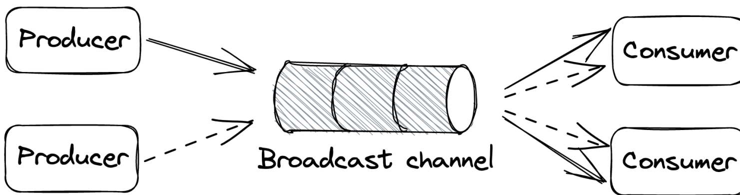

Figure 23.4: Broadcast messaging style

23.1 Guarantees

A message channel is implemented by a messaging service, or broker, like AWS SQS[1] or Kafka, which buffers messages and decouples the producer from the consumer. Different message brokers offer different guarantees depending on the tradeoffs their implementations make. For example, you would think that a channel should respect the insertion order of its messages, but you will find that some implementations, like SQS standard queues[2], don't offer any strong ordering guarantees. Why is that?

Because a message broker needs to scale horizontally just like

[1] "Amazon Simple Queue Service," https://aws.amazon.com/sqs/

[2] "Amazon SQS Standard queues," https://docs.aws.amazon.com/AWSSimpleQueueService/latest/SQSDeveloperGuide/standard-queues.html

the applications that use it, its implementation is necessarily distributed. And when multiple nodes are involved, guaranteeing order becomes challenging, as some form of coordination is required. Some brokers, like Kafka, partition a channel into multiple sub-channels. So when messages are written to the channel, they are routed to sub-channels based on their partition key. Since each partition is small enough to be handled by a single broker process, it's trivial to enforce an ordering of the messages routed to it. But to guarantee that the message order is preserved end-to-end, only a single consumer process is allowed to read from a sub-channel[3].

Because the channel is partitioned, all the caveats discussed in chapter 16 apply to it. For example, a partition could become hot enough (due to the volume of incoming messages) that the consumer reading from it can't keep up with the load. In this case, the channel might have to be rebalanced by adding more partitions, potentially degrading the broker while messages are being shuffled across partitions. It should be clear by now why not guaranteeing the order of messages makes the implementation of a broker much simpler.

Ordering is just one of the many tradeoffs a broker needs to make, such as:

- delivery guarantees, like at-most-once or at-least-once;
- message durability guarantees;
- latency;
- messaging standards supported, like AMQP[4];
- support for competing consumer instances;
- broker limits, such as the maximum supported size of messages.

Because there are so many different ways to implement channels, in the rest of this section, we will make some assumptions for the sake of simplicity:

[3] This is also referred to as the *competing consumer pattern*, which is implemented using leader election.

[4] "Advanced Message Queuing Protocol," https://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol

- Channels are point-to-point and support many producer and consumer instances.
- Messages are delivered to the consumer at least once.
- While a consumer instance is processing a message, the message remains in the channel, but other instances can't read it for the duration of a visibility timeout. The *visibility timeout* guarantees that if the consumer instance crashes while processing the message, the message will become visible to other instances when the timeout triggers. When the consumer instance is done processing the message, it deletes it from the channel, preventing it from being received by any other consumer instance in the future.

The above guarantees are similar to the ones offered by managed services such as Amazon's SQS and Azure Queue Storage[5].

23.2 Exactly-once processing

As mentioned before, a consumer instance has to delete a message from the channel once it's done processing it so that another instance won't read it. If the consumer instance deletes the message before processing it, there is a risk it could crash after deleting the message but before processing it, causing the message to be lost for good. On the other hand, if the consumer instance deletes the message only after processing it, there is a risk that it crashes after processing the message but before deleting it, causing the same message to be read again later on.

Because of that, there is no such thing[6] as *exactly-once message delivery*. So the best a consumer can do is to simulate *exactly-once message processing* by requiring messages to be idempotent and deleting them from the channel only after they have been processed.

[5] "Azure Queue Storage," https://azure.microsoft.com/en-us/services/storage/queues/

[6] "You Cannot Have Exactly-Once Delivery Redux," https://bravenewgeek.com/you-cannot-have-exactly-once-delivery-redux/

23.3 Failures

When a consumer instance fails to process a message, the visibility timeout triggers, and the message is eventually delivered to another instance. What happens if processing a specific message consistently fails with an error, though? To guard against the message being picked up repeatedly in perpetuity, we need to limit the maximum number of times the same message can be read from the channel.

To enforce a maximum number of retries, the broker can stamp messages with a counter that keeps track of the number of times the message has been delivered to a consumer. If the broker doesn't support this functionality out of the box, the consumer can implement it.

Once we have a way to count the number of times a message has been retried, we still have to decide what to do when the maximum is reached. A consumer shouldn't delete a message without processing it, as that would cause data loss. But what it can do is remove the message from the channel after writing it to a *dead letter channel* — a channel that buffers messages that have been retried too many times.

This way, messages that consistently fail are not lost forever but merely put on the side so that they don't pollute the main channel, wasting the consumer's resources. A human can then inspect these messages to debug the failure, and once the root cause has been identified and fixed, move them back to the main channel to be reprocessed.

23.4 Backlogs

One of the main advantages of using a message broker is that it makes the system more robust to outages. This is because the producer can continue writing messages to a channel even if the consumer is temporarily degraded or unavailable. As long as the arrival rate of messages is lower than or equal to their deletion rate,

everything is great. However, when that is no longer true and the consumer can't keep up with the producer, a backlog builds up.

A messaging channel introduces a bimodal behavior in the system. In one mode, there is no backlog, and everything works as expected. In the other, a backlog builds up, and the system enters a degraded state. The issue with a backlog is that the longer it builds up, the more resources and/or time it will take to drain it.

There are several reasons for backlogs, for example:

- more producer instances come online, and/or their throughput increases, and the consumer can't keep up with the arrival rate;
- the consumer's performance has degraded and messages take longer to be processed, decreasing the deletion rate;
- the consumer fails to process a fraction of the messages, which are picked up again until they eventually end up in the dead letter channel. This wastes the consumer's resources and delays the processing of healthy messages.

To detect and monitor backlogs, we can measure the average time a message waits in the channel to be read for the first time. Typically, brokers attach a timestamp of when the message was first written to it. The consumer can use that timestamp to compute how long the message has been waiting in the channel by comparing it to the timestamp taken when the message was read. Although the two timestamps have been generated by two physical clocks that aren't perfectly synchronized (see section 8.1), this measure generally provides a good warning sign of backlogs.

23.5 Fault isolation

A single producer instance that emits "poisonous" messages that repeatedly fail to be processed can degrade the consumer and potentially create a backlog because these messages are processed multiple times before they end up in the dead-letter channel. Therefore, it's important to find ways to deal with poisonous messages before that happens.

If messages are decorated with an identifier of the source that generated them, the consumer can treat them differently. For example, suppose messages from a specific user fail consistently. In that case, the consumer could decide to write these messages to an alternate low-priority channel and remove them from the main channel without processing them. The consumer reads from the slow channel but does so less frequently than the main channel, isolating the damage a single bad user can inflict to the others.

Summary

Building scalable applications boils down to exploiting three orthogonal patterns:

- breaking the application into separate services, each with its own well-defined responsibility (*functional decomposition*);
- splitting data into partitions and distributing them across nodes (*partitioning*);
- replicating functionality or data across nodes (*replication*).

We have seen plenty of applications of these patterns over the past few chapters. By now, you should have a feel for the pros and cons of each pattern.

There is another message I subtly tried to convey: there is a small subset of managed services that you can use to build a surprisingly large number of applications. The main attraction of managed services is that someone else gets paged for them.[7]

Depending on which cloud provider you are using, the name of the services and their APIs differ somewhat, but conceptually they serve the same use cases. You should be familiar with some way to run your application instances in the cloud (e.g., EC2) and load-balance traffic to them (e.g., ELB). And since you want your applications to be stateless, you also need to be familiar with a file store (e.g., S3), a key-value/document store (e.g., DynamoDB), and a messaging service (e.g., SQS, Kinesis). I would argue that these technologies are reasonable enough defaults for building a large

[7] As we will discuss in Part V, maintaining a service is no small feat.

number of scalable applications. Once you have a scalable core, you can add optimizations, such as caching in the form of managed Redis/Memcached or CDNs.

Part IV

Resiliency

Introduction

> *"Anything that can go wrong will go wrong."*
>
> – Murphy's law

In the last part, we discussed the three fundamental scalability patterns: functional decomposition, data partitioning, and replication. They all have one thing in common: they increase the number of moving parts (machines, services, processes, etc.) in our applications. But since every part has a probability of failing, the more moving parts there are, the higher the chance that any of them will fail. Eventually, anything that can go wrong will go wrong [8]; power outages, hardware faults, software crashes, memory leaks — you name it.

Remember when we talked about availability and "nines" in chapter 1? Well, to guarantee just two nines, an application can only be unavailable for up to 15 minutes a day. That's very little time to take any manual action when something goes wrong. And if we strive for three nines, we only have 43 minutes per *month* available. Clearly, the more nines we want, the faster our systems need to detect, react to, and repair faults as they occur. In this part, we will discuss a variety of resiliency best practices and patterns to achieve that.

To build fault-tolerant applications, we first need to have an idea of what can go wrong. In chapter 24, we will explore some of the most common root causes of failures.

[8] Also known as Murphy's law

Chapter 25 describes how to use redundancy, the replication of functionality or state, to increase the availability of a system. As we will learn, redundancy is only helpful when the redundant nodes can't fail for the same reason at the same time, i.e., when failures are not correlated.

Chapter 26 discusses how to isolate correlated failures by partitioning resources and then describes two very powerful forms of partitioning: shuffle sharding and cellular architectures.

Chapter 27 dives into more tactical resiliency patterns for tolerating failures of downstream dependencies that you can apply to existing systems with few changes, like timeouts and retries.

Chapter 28 discusses resiliency patterns, like load shedding and rate-limiting, for protecting systems against overload from upstream dependencies.

Chapter 24

Common failure causes

We say that a system has a *failure*[1] when it no longer provides a service to its users that meets its specification. A failure is caused by a *fault*: a failure of an internal component or an external dependency the system depends on. Some faults can be tolerated and have no user-visible impact at all, while others lead to failures.

To build fault-tolerant applications, we first need to have an idea of what can go wrong. In the next few sections, we will explore some of the most common root causes of failures. By the end of it, you will likely wonder how to tolerate all these different types of faults. The answers will follow in the next few chapters.

24.1 Hardware faults

Any physical part of a machine can fail. HDDs, memory modules, power supplies, motherboards, SSDs, NICs, or CPUs, can all stop working for various reasons. In some cases, hardware faults can cause data corruption as well. If that wasn't enough, entire data centers can go down because of power cuts or natural disasters.

[1] "A Conceptual Framework for System Fault Tolerance," https://resources.sei.cmu.edu/asset_files/TechnicalReport/1992_005_001_16112.pdf

As we will discuss later, we can address many of these infrastructure faults with redundancy. You would think that these faults are the main cause for distributed applications failing, but in reality, they often fail for very mundane reasons.

24.2 Incorrect error handling

A study from 2014[2] of user-reported failures from five popular distributed data stores found that the majority of catastrophic failures were the result of incorrect handling of non-fatal errors.

In most cases, the bugs in the error handling could have been detected with simple tests. For example, some handlers completely ignored errors. Others caught an overly generic exception, like *Exception* in Java, and aborted the entire process for no good reason. And some other handlers were only partially implemented and even contained "FIXME" and "TODO" comments.

In hindsight, this is perhaps not too surprising, given that error handling tends to be an afterthought.[3] Later, in chapter 29, we will take a closer look at best practices for testing large distributed applications.

24.3 Configuration changes

Configuration changes are one of the leading root causes for catastrophic failures[4]. It's not just misconfigurations that cause problems, but also valid configuration changes to enable rarely-used features that no longer work as expected (or never did).

What makes configuration changes particularly dangerous is that their effects can be delayed[5]. If an application reads a configura-

[2] "Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems," https://www.usenix.org/system/files/conference/osdi14/osdi14-paper-yuan.pdf

[3] This is the reason the Go language puts so much emphasis on error handling.

[4] "A List of Post-mortems: Config Errors," https://github.com/danluu/post-mortems#config-errors

[5] "Early Detection of Configuration Errors to Reduce Failure Damage," https://www.usenix.org/system/files/conference/osdi16/osdi16-xu.pdf

tion value only when it's actually needed, an invalid value might take effect only hours or days after it has changed and thus escape early detection.

This is why configuration changes should be version-controlled, tested, and released just like code changes, and their validation should happen preventively when the change happens. In chapter 30, we will discuss safe release practices for code and configuration changes in the context of continuous deployments.

24.4 Single points of failure

A single point of failure (SPOF) is a component whose failure brings the entire system down with it. In practice, systems can have multiple SPOFs.

Humans make for great SPOFs, and if you put them in a position where they can cause a catastrophic failure on their own, you can bet they eventually will. For example, human failures often happen when someone needs to manually execute a series of operational steps in a specific order without making any mistakes. On the other hand, computers are great at executing instructions, which is why automation should be leveraged whenever possible.

Another common SPOF is DNS[6]. If clients can't resolve the domain name for an application, they won't be able to connect to it. There are many reasons why that can happen, ranging from domain names expiring[7] to entire root level domains going down[8].

Similarly, the TLS certificate used by an application for its HTTP endpoints is also a SPOF[9]. If the certificate expires, clients won't be able to open a secure connection with the application.

[6] "It's always DNS," https://twitter.com/ahidalgosre/status/1315345619926609920?lang=en-GB

[7] "Foursquare Goes Dark Too. Unintentionally.," https://techcrunch.com/2010/03/27/foursquare-offline

[8] "Stop using .IO Domain Names for Production Traffic," https://hackernoon.com/stop-using-io-domain-names-for-production-traffic-b6aa17eeac20

[9] "Microsoft Teams goes down after Microsoft forgot to renew a certificate," https://www.theverge.com/2020/2/3/21120248/microsoft-teams-down-outage-certificate-issue-status

Ideally, SPOFs should be identified when the system is designed. The best way to detect them is to examine every system component and ask what would happen if it were to fail. Some SPOFs can be architected away, e.g., by introducing redundancy, while others can't. In that case, the only option left is to reduce the SPOF's blast radius, i.e., the damage the SPOF inflicts on the system when it fails. Many of the resiliency patterns we will discuss later reduce the blast radius of failures.

24.5 Network faults

When a client sends a request to a server, it expects to receive a response from it a while later. In the best case, it receives the response shortly after sending the request. If that doesn't happen, the client has two options: continue to wait or fail the request with a time-out exception or error.

As discussed in chapter 7, when the concepts of failure detection and timeouts were introduced, there are many reasons for not getting a prompt response. For example, the server could be very slow or have crashed while processing the request; or maybe the network could be losing a small percentage of packets, causing lots of retransmissions and delays.

Slow network calls are the silent killers[10] of distributed systems. Because the client doesn't know whether the response will eventually arrive, it can spend a long time waiting before giving up, if it gives up at all, causing performance degradations that are challenging to debug. This kind of fault is also referred to as a *gray failure*[11]: a failure that is so subtle that it can't be detected quickly or accurately. Because of their nature, gray failures can easily bring an entire system down to its knees.

In the next section, we will explore another common cause of gray failures.

[10] "Fallacies of distributed computing," https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

[11] "Gray Failure: The Achilles' Heel of Cloud-Scale Systems," https://www.microsoft.com/en-us/research/wp-content/uploads/2017/06/paper-1.pdf

24.6 Resource leaks

From an observer's point of view, a very slow process is not very different from one that isn't running at all — neither can perform useful work. Resource leaks are one of the most common causes of slow processes.

Memory is arguably the most well-known resource affected by leaks. A memory leak causes a steady increase in memory consumption over time. Even languages with garbage collection are vulnerable to leaks: if a reference to an object that is no longer needed is kept somewhere, the garbage collector won't be able to delete it. When a leak has consumed so much memory that there is very little left, the operating system will start to swap memory pages to disk aggressively. Also, the garbage collector will kick in more frequently, trying to release memory. All of this consumes CPU cycles and makes the process slower. Eventually, when there is no more physical memory left, and there is no more space in the swap file, the process won't be able to allocate memory, and most operations will fail.

Memory is just one of the many resources that can leak. Take thread pools, for example: if a thread acquired from a pool makes a synchronous blocking HTTP call without a timeout and the call never returns, the thread won't be returned to the pool. And since the pool has a limited maximum size, it will eventually run out of threads if it keeps losing them.

You might think that making *asynchronous* calls rather than synchronous ones would help in the previous case. However, modern HTTP clients use socket pools to avoid recreating TCP connections and paying a performance fee, as discussed in chapter 2. If a request is made without a timeout, the connection is never returned to the pool. As the pool has a limited maximum size, eventually, there won't be any connections left.

On top of that, your code isn't the only thing accessing memory, threads, and sockets. The libraries your application depends on use the same resources, and they can hit the same issues we just

discussed.

24.7 Load pressure

Every system has a limit of how much load it can withstand, i.e., its capacity. So when the load directed to the system continues to increase, it's bound to hit that limit sooner or later. But an organic increase in load, that gives the system the time to scale out accordingly and increase its capacity, is one thing, and a sudden and unexpected flood is another.

For example, consider the number of requests received by an application in a period of time. The rate and the type of incoming requests can change over time, and sometimes suddenly, for a variety of reasons:

- The requests might have a seasonality. So, for example, depending on the hour of the day, the application is hit by users in different countries.
- Some requests are much more expensive than others and abuse the system in unexpected ways, like scrapers slurping in data at super-human speed.
- Some requests are malicious, like those of DDoS attacks that try to saturate the application's bandwidth to deny legitimate users access to it.

While some load surges can be handled by automation that adds capacity (e.g., autoscaling), others require the system to reject requests to shield it from overloading, using the patterns we will discuss in chapter 28.

24.8 Cascading failures

You would think that if a system has hundreds of processes, it shouldn't make much of a difference if a small percentage are slow or unreachable. The thing about faults is that they have the potential to spread virally and cascade from one process to the other until the whole system crumbles to its knees. This happens when

system components depend on each other, and a failure in one increases the probability of failure in others.

For example, suppose multiple clients are querying two database replicas, A and B, behind a load balancer. Each replica handles about 50 transactions per second (see Figure 24.1).

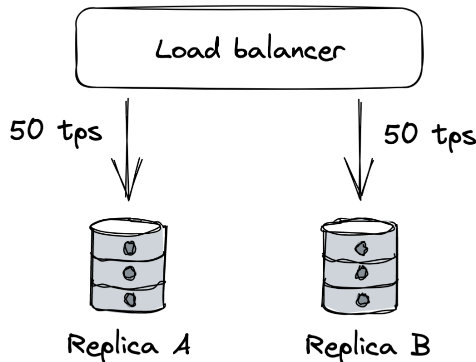

Figure 24.1: Two replicas behind a load balancer; each is handling half the load.

Suddenly, replica B becomes unavailable because of a network fault. The load balancer detects that B is unavailable and removes it from the pool. Because of that, replica A has to pick up the slack for B and serve twice the requests per unit time it was serving before (see Figure 24.2).

If replica A struggles to keep up with the incoming requests, the clients will experience increasingly more timeouts and start to retry requests, adding more load to the system. Eventually, replica A will be under so much load that most requests will time out, forcing the load balancer to remove it from the pool. In other words, the original network fault that caused replica B to become unavailable cascaded into a fault at replica A.

Suppose now that replica B becomes available again and the load balancer puts it back in the pool. Because it's the only replica in the pool, it will be flooded with requests, causing it to overload and eventually be removed again.

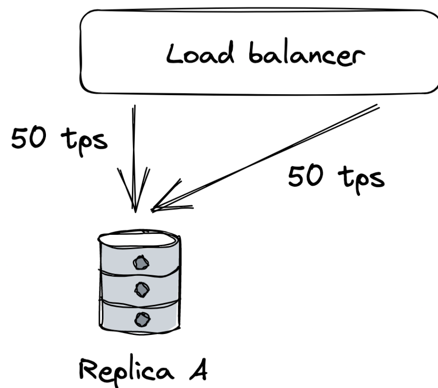

Figure 24.2: When replica B becomes unavailable, A will be hit with more load, which can strain it beyond its capacity.

You can see how even after the network fault is gone, the application continues to struggle because of a feedback loop that causes the load to jump from one replica to another. Failures with this characteristic are also referred to as metastable[12].

A big enough corrective action is usually needed to break the loop, like temporarily blocking traffic from getting to the replicas in the first place. Unfortunately, these failures are very hard to mitigate once they have started, and the best way to prevent them is to stop faults from spreading from one component to another in the first place.

24.9 Managing risk

As it should be evident by now, a distributed application needs to accept that faults are inevitable and be prepared to detect, react to, and repair them as they occur.

At this point, you might feel overwhelmed by the sheer amount of things that can go wrong. But just because a specific fault has

[12] "Metastable Failures in Distributed Systems," https://sigops.org/s/conferences/hotos/2021/papers/hotos21-s11-bronson.pdf

a chance of happening, it doesn't mean we have to do something about it. We first have to consider the probability it will manifest and the impact it will cause to the system's users when it does. By multiplying the two factors together, we get a risk score[13] that we can use to prioritize which faults to address (see Figure 24.3) first. For example, a fault that is very likely to happen, and has a large impact, should be tackled head on; on the other hand, a fault with a low likelihood and low impact can wait.

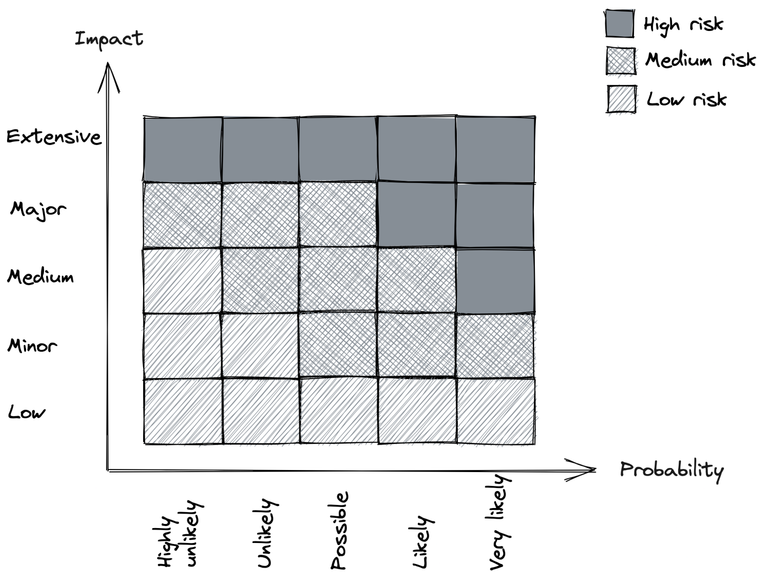

Figure 24.3: Risk matrix

Once we decide that we need to do something about a specific fault, we can try to reduce its probability and/or reduce its impact. This will be the main focus of the next chapters.

[13] "Risk matrix," https://en.wikipedia.org/wiki/Risk_matrix

Chapter 25

Redundancy

Redundancy, the replication of functionality or state, is arguably the first line of defense against failures. When functionality or state is replicated over multiple nodes and a node fails, the others can take over. Moreover, as discussed in Part III, replication is also a core pattern that enables our applications to scale out horizontally.

Redundancy is the main reason why distributed applications can achieve better availability than single-node applications. But only some forms of redundancy actually improve availability. Marc Brooker lists four prerequisites[1]:

1. The complexity added by introducing redundancy mustn't cost more availability than it adds.
2. The system must reliably detect which of the redundant components are healthy and which are unhealthy.
3. The system must be able to run in degraded mode.
4. The system must be able to return to fully redundant mode.

Let's see how these prerequisites apply to a concrete example. Hardware faults such as disk, memory, and network failures can cause a node to crash, degrade or become otherwise unavailable. In a stateless service, a load balancer can mask these faults using

[1] "When Redundancy Actually Helps," https://brooker.co.za/blog/2019/06/20/redundancy.html

a pool of redundant nodes. Although the load balancer increases the system's complexity and, therefore, the number of ways the system can fail, the benefits in terms of scalability and availability almost always outweigh the risks it introduces (e.g., the load balancer failing).

The load balancer needs to detect which nodes are healthy and which aren't to take the faulty ones out of the pool. It does that with health checks, as we learned in chapter 18. Health checks are critical to achieving high availability; if there are ten servers in the pool and one is unresponsive for some reason, then 10% of requests will fail, causing the availability to drop. Therefore, the longer it takes for the load balancer to detect the unresponsive server, the longer the failures will be visible to the clients.

Now, when the load balancer takes one or more unhealthy servers out of the pool, the assumption is that the others have enough capacity left to handle the increase in load. In other words, the system must be able to run in degraded mode. However, that by itself is not enough; new servers also need to be added to the pool to replace the ones that have been removed. Otherwise, there eventually won't be enough servers left to cope with the load.

In stateful services, masking a node failure is a lot more complex since it involves replicating state. We have discussed replication at length in the previous chapters, and it shouldn't come as a surprise by now that meeting the above requisites is a lot more challenging for a stateful service than for a stateless one.

25.1 Correlation

Redundancy is only helpful when the redundant nodes can't fail for the same reason at the same time, i.e., when failures are not correlated. For example, if a faulty memory module causes a server to crash, it's unlikely other servers will fail simultaneously for the same reason since they are running on different machines. However, if the servers are hosted in the same data center, and a fiber cut or an electrical storm causes a data-center-wide outage, the en-

tire application becomes unavailable no matter how many servers there are. In other words, the failures caused by a data center outage are correlated and limit the application's availability. So if we want to increase the availability, we have to reduce the correlation between failures by using more than one data center.

Cloud providers such as AWS and Azure replicate their entire stack in multiple regions for that very reason. Each region comprises multiple data centers called Availability Zones (AZs) that are cross-connected with high-speed network links. AZs are far enough from each other to minimize the risk of correlated failures (e.g., power cuts) but still close enough to have low network latency, which is bounded by the speed of light. In fact, the latency is low enough by design to support synchronous replication protocols without a significant latency penalty.

With AZs, we can create applications that are resilient to data center outages. For example, a stateless service could have instances running in multiple AZs behind a shared load balancer so that if an AZ becomes unavailable, it doesn't impact the availability of the service. On the other hand, stateful services require the use of a replication protocol to keep their state in sync across AZs. But since latencies are low enough between AZs, the replication protocol can be partially synchronous, like Raft, or even fully synchronous, like chain replication.

Taking it to the extreme, a catastrophic event could destroy an entire region with all of its AZs. To tolerate that, we can duplicate the entire application stack in multiple regions. To distribute the traffic to different data centers located in different regions, we can use *global DNS load balancing*. Unlike earlier, the application's state needs to be replicated asynchronously across regions[2] given the high network latency between regions (see Figure 25.1).

That said, the chance of an entire region being destroyed is extremely low. Before embarking on the effort of making your application resilient against region failures, you should have very good

[2] "Active-Active for Multi-Regional Resiliency," https://netflixtechblog.com/active-active-for-multi-regional-resiliency-c47719f6685b

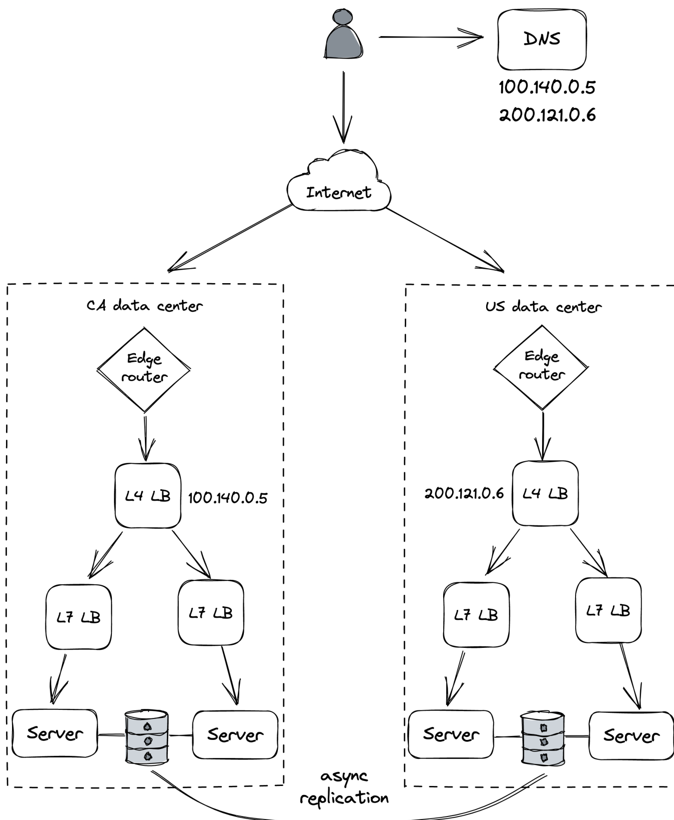

Figure 25.1: A simplistic multi-region architecture

reasons for it. It's more likely your application will be forced to have a presence in multiple regions for legal compliance reasons. For example, there are laws mandating that the data of European customers has to be processed and stored within Europe[3].

[3]"The CJEU judgment in the Schrems II case," https://www.europarl.europa.eu/RegData/etudes/ATAG/2020/652073/EPRS_ATA(2020)652073_EN.pdf

Chapter 26

Fault isolation

So far, we have discussed how to address infrastructure faults with redundancy, but there are other kinds of failures that we can't tolerate with redundancy alone because of their high degree of correlation.

For example, suppose a specific user sends malformed requests (deliberately or not) that cause the servers handling them to crash because of a bug. Since the bug is in the code, it doesn't matter how many DCs and regions our application is deployed to; if the user's requests can land anywhere, they can affect all DCs and regions. Due to their nature, these requests are sometimes referred to as poison pills.

Similarly, if the requests of a specific user require a lot more resources than others, they can degrade the performance for every other user (aka noisy neighbor effect).

The main issue in the previous examples is that the blast radius of poison pills and noisy neighbors is the entire application. To reduce it, we can partition the application's stack by user so that the requests of a specific user can only ever affect the partition it was assigned to.[1] That way, even if a user is degrading a partition,

[1]We discussed partitioning in chapter 16 from a scalability point of view.

the issue is isolated from the rest of the system.

For example, suppose we have 6 instances of a stateless service behind a load balancer, divided into 3 partitions (see Figure 26.1). In this case, a noisy or poisonous user can only ever impact 33 percent of users. And as the number of partitions increases, the blast radius decreases further.

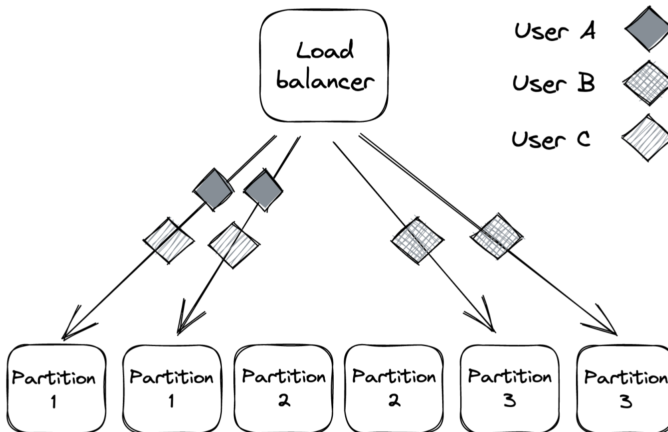

Figure 26.1: Service instances partitioned into 3 partitions

The use of partitions for fault isolation is also referred to as the *bulkhead pattern*, named after the compartments of a ship's hull. If one compartment is damaged and fills up with water, the leak is isolated to that partition and doesn't spread to the rest of the ship.

26.1 Shuffle sharding

The problem with partitioning is that users who are unlucky enough to land on a degraded partition are impacted as well. For stateless services, there is a very simple, yet powerful, variation of partitioning called shuffle sharding[2] that can help mitigate that.

The idea is to introduce *virtual partitions* composed of random (but

[2] "Shuffle Sharding: Massive and Magical Fault Isolation," https://aws.amazon.com/blogs/architecture/shuffle-sharding-massive-and-magical-fault-isolation/

permanent) subsets of service instances. This makes it much more unlikely for two users to be allocated to the same partition as each other.

Let's go back to our previous example of a stateless service with 6 instances. How many combinations of virtual partitions with 2 instances can we build out of 6 instances? If you recall the combinations formula from your high school statistics class, the answer is 15:

$$\frac{n!}{r!(n-r)!} = \frac{6!}{2!4!} = 15$$

There are now 15 partitions for a user to be assigned to, while before, we had only 3, which makes it a lot less likely for two users to end up in the same partition. The caveat is that virtual partitions partially overlap (see Figure 26.2). But by combining shuffle sharding with a load balancer that removes faulty instances, and clients that retry failed requests, we can build a system with much better fault isolation than one with physical partitions alone.

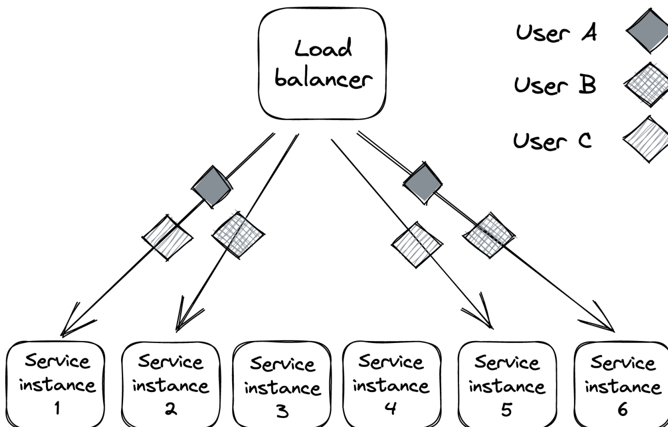

Figure 26.2: Virtual partitions are far less likely to fully overlap with each other.

26.2 Cellular architecture

In the previous examples, we discussed partitioning in the context of stateless services. We can take it up a notch and partition the entire application stack, including its dependencies (load balancers, compute services, storage services, etc.), by user[3] into *cells*[4]. Each cell is completely independent of others, and a gateway service is responsible for routing requests to the right cells.

We have already seen an example of a "cellular" architecture when discussing Azure Storage in Chapter 17. In Azure Storage, a cell is a storage cluster, and accounts are partitioned across storage clusters (see Figure 26.3).

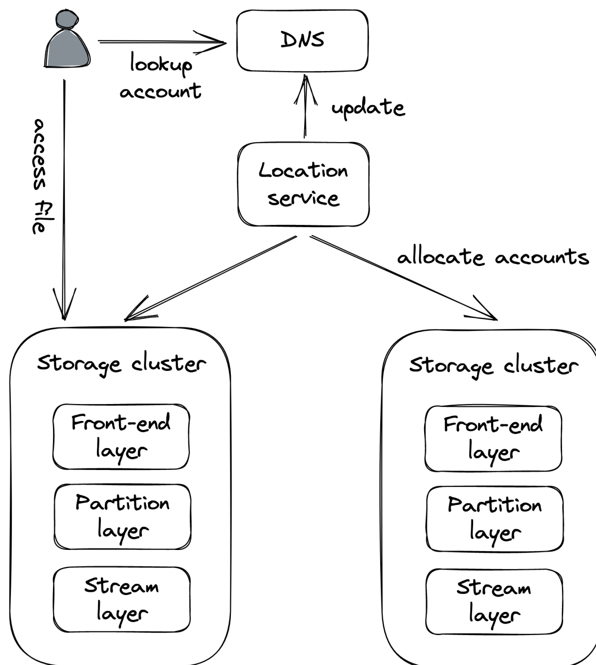

Figure 26.3: Each storage cluster (stamp) is a cell in Azure Storage.

[3]Partitioning by user is just an example; we could partition just as well by physical location, workload, or any other dimension that makes sense for the application.

[4]"New Relic case: Huge scale, small clusters: Using Cells to scale in the Cloud," https://www.youtube.com/watch?v=eMikCXiBlOA

An unexpected benefit of cellular architectures comes from setting limits to the maximum capacity of a cell. That way, when the system needs to scale out, a new cell is added rather than scaling out existing ones. Since a cell has a maximum size, we can thoroughly test and benchmark it at that size, knowing that we won't have any surprises in the future and hit some unexpected brick wall.

Chapter 27

Downstream resiliency

Now that we have discussed how to reduce the impact of faults at the architectural level with redundancy and partitioning, we will dive into tactical resiliency patterns that stop faults from propagating from one component or service to another. In this chapter, we will discuss patterns that protect a service from failures of downstream dependencies.

27.1 Timeout

When a network call is made, it's best practice to configure a timeout to fail the call if no response is received within a certain amount of time. If the call is made without a timeout, there is a chance it will never return, and as mentioned in chapter 24, network calls that don't return lead to resource leaks. Thus, the role of timeouts is to detect connectivity faults and stop them from cascading from one component to another. In general, timeouts are a must-have for operations that can potentially never return, like acquiring a mutex.

Unfortunately, some network APIs don't have a way to set a timeout in the first place, while others have no timeout configured by default. For example, JavaScript's *XMLHttpRequest* is *the* web API

to retrieve data from a server asynchronously, and its default timeout is zero[1], which means there is no timeout:

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "/api", true);
// No timeout by default, so it needs to be set explicitly!
xhr.timeout = 10000; // 10K milliseconds
xhr.onload = function () {
  // Request finished
};
xhr.ontimeout = function (e) {
  // Request timed out
};
xhr.send(null);
```

The *fetch* web API is a modern replacement for *XMLHttpRequest* that uses Promises. When the fetch API was initially introduced, there was no way to set a timeout at all[2]. Browsers have only later added support for timeouts through the Abort API[3]. Things aren't much rosier for Python; the popular *requests* library uses a default timeout of infinity[4]. And Go's *HTTP package* doesn't use timeouts[5] by default.

Modern HTTP clients for Java and .NET do a better job and usually, come with default timeouts. For example, .NET Core *HttpClient* has a default timeout of 100 seconds[6]. It's lax but arguably better than not having a timeout at all.

As a rule of thumb, always set timeouts when making network calls, and be wary of third-party libraries that make network calls

[1] "Web APIs: XMLHttpRequest.timeout," https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/timeout

[2] "Add a timeout option, to prevent hanging," https://github.com/whatwg/fetch/issues/951

[3] "Web APIs: AbortController," https://developer.mozilla.org/en-US/docs/Web/API/AbortController

[4] "Requests Quickstart: Timeouts," https://requests.readthedocs.io/en/master/user/quickstart/#timeouts

[5] "net/http: make default configs have better timeouts," https://github.com/golang/go/issues/24138

[6] "HttpClient.Timeout Property," https://docs.microsoft.com/en-us/dotnet/api/system.net.http.httpclient.timeout?view=net-6.0#remarks

but don't expose settings for timeouts.

But how do we determine a good timeout duration? One way is to base it on the desired false timeout rate[7]. For example, suppose we have a service calling another, and we are willing to accept that 0.1% of downstream requests that would have eventually returned a response time out (i.e., 0.1% false timeout rate). To accomplish that, we can configure the timeout based on the 99.9th percentile of the downstream service's response time.

We also want to have good monitoring in place to measure the entire lifecycle of a network call, like the duration of the call, the status code received, and whether a timeout was triggered. We will talk more about monitoring later in the book, but the point I want to make here is that we have to measure what happens at the integration points of our systems, or we are going to have a hard time debugging production issues.

Ideally, a network call should be wrapped within a library function that sets a timeout and monitors the request so that we don't have to remember to do this for each call. Alternatively, we can also use a reverse proxy co-located on the same machine, which intercepts remote calls made by our process. The proxy can enforce timeouts and monitor calls, relieving our process ofthis responsibility. We talked about this in section 18.3 when discussing the sidecar pattern and the service mesh.

27.2 Retry

We know by now that a client should configure a timeout when making a network request. But what should it do when the request fails or times out? The client has two options at that point: it can either fail fast or retry the request. If a short-lived connectivity issue caused the failure or timeout, then retrying after some *backoff time* has a high probability of succeeding. However, if the downstream service is overwhelmed, retrying immediately after

[7] "Timeouts, retries, and backoff with jitter," https://aws.amazon.com/builders-library/timeouts-retries-and-backoff-with-jitter/

will only worsen matters. This is why retrying needs to be slowed down with increasingly longer delays between the individual retries until either a maximum number of retries is reached or enough time has passed since the initial request.

27.2.1 Exponential backoff

To set the delay between retries, we can use a *capped exponential function*, where the delay is derived by multiplying the initial backoff duration by a constant that increases exponentially after each attempt, up to some maximum value (the cap):

$$\text{delay} = min(\text{cap}, \text{initial-backoff} \cdot 2^{\text{attempt}})$$

For example, if the cap is set to 8 seconds, and the initial backoff duration is 2 seconds, then the first retry delay is 2 seconds, the second is 4 seconds, the third is 8 seconds, and any further delay will be capped to 8 seconds.

Although exponential backoff does reduce the pressure on the downstream dependency, it still has a problem. When the downstream service is temporarily degraded, multiple clients will likely see their requests failing around the same time. This will cause clients to retry simultaneously, hitting the downstream service with load spikes that further degrade it, as shown in Figure 27.1.

To avoid this herding behavior, we can introduce random jitter[8] into the delay calculation. This spreads retries out over time, smoothing out the load to the downstream service:

$$\text{delay} = random(0, min(\text{cap}, \text{initial-backoff} \cdot 2^{\text{attempt}}))$$

Actively waiting and retrying failed network requests isn't the only way to implement retries. In batch applications that don't have strict real-time requirements, a process can park a failed

[8] "Exponential Backoff And Jitter," https://aws.amazon.com/blogs/architecture/exponential-backoff-and-jitter/

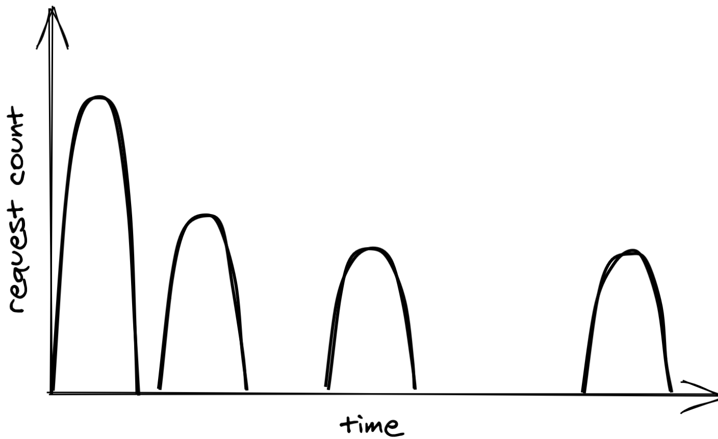

Figure 27.1: Retry storm

request into a *retry queue*. The same process, or possibly another, can read from the same queue later and retry the failed requests.

Just because a network call can be retried doesn't mean it should be. If the error is not short-lived, for example, because the process is not authorized to access the remote endpoint, it makes no sense to retry the request since it will fail again. In this case, the process should fail fast and cancel the call right away. And as discussed in chapter 5.7, we should also understand the consequences of retrying a network call that isn't idempotent and whose side effects can affect the application's correctness.

27.2.2 Retry amplification

Suppose that handling a user request requires going through a chain of three services. The user's client calls service A, which calls service B, which in turn calls service C. If the intermediate request from service B to service C fails, should B retry the request or not? Well, if B does retry it, A will perceive a longer execution time for its request, making it more likely to hit A's timeout. If that happens, A retries the request, making it more likely for the client to hit its timeout and retry.

Having retries at multiple levels of the dependency chain can amplify the total number of retries — the deeper a service is in the chain, the higher the load it will be exposed to due to retry amplification (see Figure 27.2).

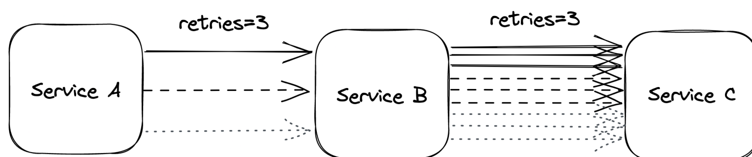

Figure 27.2: Retry amplification in action

And if the pressure gets bad enough, this behavior can easily overload downstream services. That's why, when we have long dependency chains, we should consider retrying at a single level of the chain and failing fast in all the others.

27.3 Circuit breaker

Suppose a service uses timeouts to detect whether a downstream dependency is unavailable and retries to mitigate transient failures. If the failures aren't transient and the downstream dependency remains unresponsive, what should it do then? If the service keeps retrying failed requests, it will necessarily become slower for its clients. In turn, this slowness can spread to the rest of the system.

To deal with non-transient failures, we need a mechanism that detects long-term degradations of downstream dependencies and stops new requests from being sent downstream in the first place. After all, the fastest network call is the one we don't have to make. The mechanism in question is the *circuit breaker*, inspired by the same functionality implemented in electrical circuits.

The goal of the circuit breaker is to allow a sub-system to fail without slowing down the caller. To protect the system, calls to the failing sub-system are temporarily blocked. Later, when the sub-system recovers and failures stop, the circuit breaker allows calls

to go through again.

Unlike retries, circuit breakers prevent network calls entirely, making the pattern particularly useful for non-transient faults. In other words, retries are helpful when the expectation is that the next call will succeed, while circuit breakers are helpful when the expectation is that the next call will fail.

A circuit breaker can be implemented as a state machine with three states: open, closed, and half-open (see Figure 27.3).

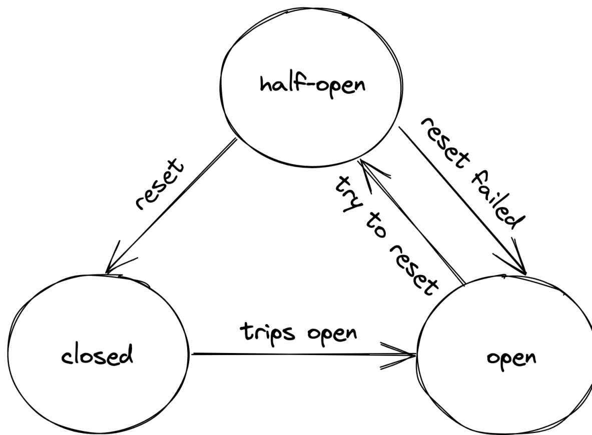

Figure 27.3: Circuit breaker state machine

In the closed state, the circuit breaker merely acts as a pass-through for network calls. In this state, the circuit breaker tracks the number of failures, like errors and timeouts. If the number goes over a certain threshold within a predefined time interval, the circuit breaker trips and opens the circuit.

When the circuit is open, network calls aren't attempted and fail immediately. As an open circuit breaker can have business implications, we need to consider what should happen when a downstream dependency is down. If the dependency is non-critical, we want our service to degrade gracefully rather than to stop entirely. Think of an airplane that loses one of its non-critical sub-systems in flight; it shouldn't crash but rather gracefully degrade to a state

where the plane can still fly and land. Another example is Amazon's front page; if the recommendation service is unavailable, the page renders without recommendations. It's a better outcome than failing to render the whole page entirely.

After some time has passed, the circuit breaker gives the downstream dependency another chance and transitions to the half-open state. In the half-open state, the next call is allowed to pass through to the downstream service. If the call succeeds, the circuit breaker transitions to the closed state; if the call fails instead, it transitions back to the open state.

You might think that's all there is to understand how a circuit breaker works, but the devil is in the details. For example, how many failures are enough to consider a downstream dependency down? How long should the circuit breaker wait to transition from the open to the half-open state? It really depends on the specific context; only by using data about past failures can we make an informed decision.

Chapter 28

Upstream resiliency

The previous chapter discussed patterns that protect services against downstream failures, like failures to reach an external dependency. In this chapter, we will shift gears and discuss mechanisms to protect against upstream pressure.[1]

28.1 Load shedding

A server has very little control over how many requests it receives at any given time. The operating system has a connection queue per port with a limited capacity that, when reached, causes new connection attempts to be rejected immediately. But typically, under extreme load, the server crawls to a halt before that limit is reached as it runs out of resources like memory, threads, sockets, or files. This causes the response time to increase until eventually, the server becomes unavailable to the outside world.

When a server operates at capacity, it should reject excess requests[2] so that it can dedicate its resources to the requests it's already processing. For example, the server could use a counter to measure

[1] We have already met a mechanism to protect against upstream pressure when we discussed health checks in the context of load balancers in chapter 18.

[2] "Using load shedding to avoid overload," https://aws.amazon.com/builders-library/using-load-shedding-to-avoid-overload

the number of concurrent requests being processed that is incremented when a new request comes in and decreased when a response is sent. The server can then infer whether it's overloaded by comparing the counter with a threshold that approximates the server's capacity.

When the server detects that it's overloaded, it can reject incoming requests by failing fast and returning a response with status code *503 (Service Unavailable)*. This technique is also referred to as *load shedding*. The server doesn't necessarily have to reject arbitrary requests; for example, if different requests have different priorities, the server could reject only low-priority ones. Alternatively, the server could reject the oldest requests first since those will be the first ones to time out and be retried, so handling them might be a waste of time.

Unfortunately, rejecting a request doesn't completely shield the server from the cost of handling it. Depending on how the rejection is implemented, the server might still have to pay the price of opening a TLS connection and reading the request just to reject it. Hence, load shedding can only help so much, and if load keeps increasing, the cost of rejecting requests will eventually take over and degrade the server.

28.2 Load leveling

There is an alternative to load shedding, which can be exploited when clients don't expect a prompt response. The idea is to introduce a messaging channel between the clients and the service. The channel decouples the load directed to the service from its capacity, allowing it to process requests at its own pace.

This pattern is referred to as load leveling and it's well suited to fending off short-lived spikes, which the channel smooths out (see Figure 28.1). But if the service doesn't catch up eventually, a large backlog will build up, which comes with its own problems, as discussed in chapter 23.

Load-shedding and load leveling don't address an increase in load

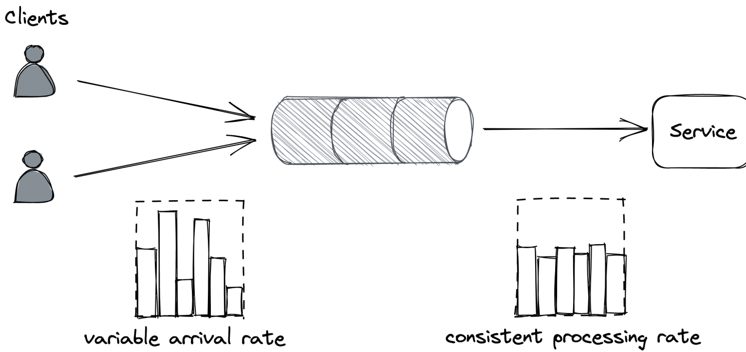

Figure 28.1: The channel smooths out the load for the consuming service.

directly but rather protect a service from getting overloaded. To handle more load, the service needs to be scaled out. This is why these protection mechanisms are typically combined with auto-scaling[3], which detects that the service is running hot and automatically increases its scale to handle the additional load.

28.3 Rate-limiting

Rate-limiting, or throttling, is a mechanism that rejects a request when a specific quota is exceeded. A service can have multiple quotas, e.g., for the number of requests or bytes received within a time interval. Quotas are typically applied to specific users, API keys, or IP addresses.

For example, if a service with a quota of 10 requests per second per API key receives on average 12 requests per second from a specific API key, it will, on average, reject 2 requests per second from that API key.

When a service rate-limits a request, it needs to return a response with a particular error code so that the sender knows that it failed because a quota has been exhausted. For services with HTTP APIs,

[3]"Autoscaling," https://en.wikipedia.org/wiki/Autoscaling

the most common way to do that is by returning a response with status code *429 (Too Many Requests)*. The response should include additional details about which quota has been exhausted and by how much; it can also include a *Retry-After* header indicating how long to wait before making a new request:

If the client application plays by the rules, it will stop hammering the service for some time, shielding the service from non-malicious users monopolizing it by mistake. In addition, this protects against bugs in the clients that cause a client to hit a downstream service for one reason or another repeatedly.

Rate-limiting is also used to enforce pricing tiers; if users want to use more resources, they should also be willing to pay more. This is how you can offload your service's cost to your users: have them pay proportionally to their usage and enforce pricing tiers with quotas.

You would think that rate-limiting also offers strong protection against a DDoS attack, but it only partially protects a service from it. Nothing forbids throttled clients from continuing to hammer a service after getting *429s*. Rate-limited requests aren't free either — for example, to rate-limit a request by API key, the service has to pay the price of opening a TLS connection, and at the very least, download part of the request to read the key. Although rate limiting doesn't fully protect against DDoS attacks, it does help reduce their impact.

Economies of scale are the only true protection against DDoS attacks. If you run multiple services behind one large gateway service, no matter which of the services behind it are attacked, the gateway service will be able to withstand the attack by rejecting the traffic upstream. The beauty of this approach is that the cost of running the gateway is amortized across all the services that are using it.

Although rate-limiting has some similarities with load shedding, they are different concepts. Load shedding rejects traffic based on the local state of a process, like the number of requests concurrently processed by it; rate-limiting instead sheds traffic based on

the global state of the system, like the total number of requests concurrently processed for a specific API key across all service instances. And because there is a global state involved, some form of coordination is required.

28.3.1 Single-process implementation

The distributed implementation of rate-limiting is interesting in its own right, and it's well worth spending some time discussing it. We will start with a single-process implementation first and then extend it to a distributed one.

Suppose we want to enforce a quota of 2 requests per minute, per API key. A naive approach would be to use a doubly-linked list per API key, where each list stores the timestamps of the last N requests received. Whenever a new request comes in, an entry is appended to the list with its corresponding timestamp. Then, periodically, entries older than a minute are purged from the list.

By keeping track of the list's length, the process can rate-limit incoming requests by comparing it with the quota. The problem with this approach is that it requires a list per API key, which quickly becomes expensive in terms of memory as it grows with the number of requests received.

To reduce memory consumption, we need to come up with a way to reduce the storage requirements. One way to do this is by dividing time into buckets of fixed duration, for example of 1 minute, and keeping track of how many requests have been seen within each bucket (see Figure 28.2).

A bucket contains a numerical counter. When a new request comes in, its timestamp is used to determine the bucket it belongs to. For example, if a request arrives at 12.00.18, the counter of the bucket for minute "12.00" is incremented by 1 (see Figure 28.3).

With bucketing, we can compress the information about the number of requests seen in a way that doesn't grow with the number of requests. Now that we have a memory-friendly representation, how can we use it to implement rate-limiting? The idea is to use a

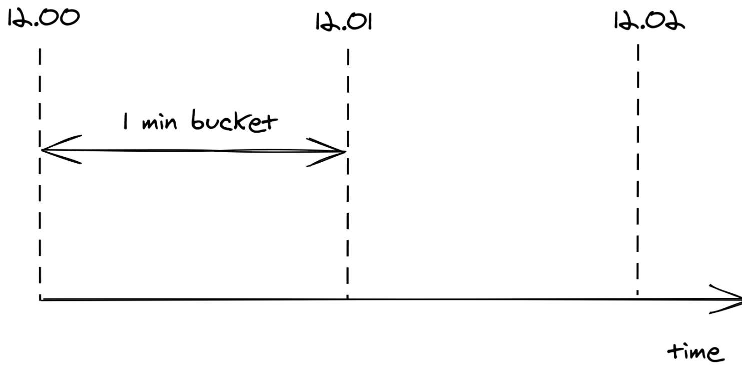

Figure 28.2: Buckets divide time into 1-minute intervals, which keep track of the number of requests seen.

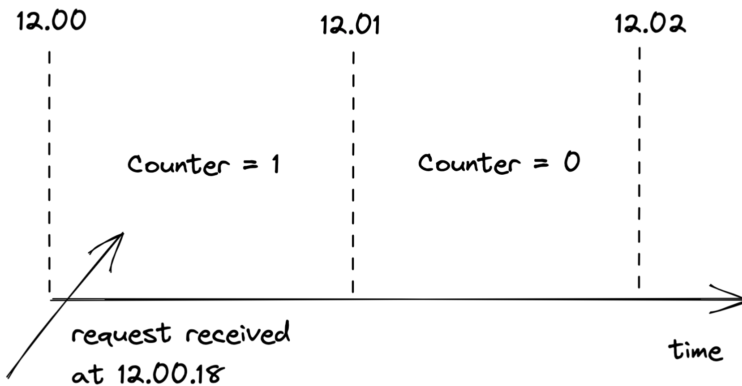

Figure 28.3: When a new request comes in, its timestamp is used to determine the bucket it belongs to.

sliding window that moves across the buckets in real time, keeping track of the number of requests within it.

The sliding window represents the interval of time used to decide whether to rate-limit or not. The window's length depends on the time unit used to define the quota, which in our case is 1 minute. But there is a caveat: a sliding window can overlap with multiple

buckets. To derive the number of requests under the sliding window, we have to compute a weighted sum of the bucket's counters, where each bucket's weight is proportional to its overlap with the sliding window (see Figure 28.4).

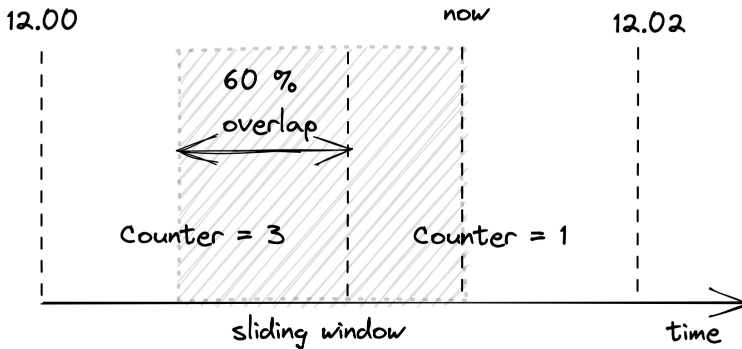

Figure 28.4: A bucket's weight is proportional to its overlap with the sliding window.

Although this is an approximation, it's a reasonably good one for our purposes. And it can be made more accurate by increasing the granularity of the buckets. So, for example, we can reduce the approximation error using 30-second buckets rather than 1-minute ones.

We only have to store as many buckets as the sliding window can overlap with at any given time. For example, with a 1-minute window and a 1-minute bucket length, the sliding window can overlap with at most 2 buckets. Thus, there is no point in storing the third oldest bucket, the fourth oldest one, etc.

To summarize, this approach requires two counters per API key, which is much more efficient in terms of memory than the naive implementation storing a list of requests per API key.

28.3.2 Distributed implementation

When more than one process accepts requests, the local state is no longer good enough, as the quota needs to be enforced on the total

number of requests per API key across all service instances. This requires a shared data store to keep track of the number of requests seen.

As discussed earlier, we need to store two integers per API key, one for each bucket. When a new request comes in, the process receiving it could fetch the current bucket, update it and write it back to the data store. But that wouldn't work because two processes could update the same bucket concurrently, which would result in a lost update. The fetch, update, and write operations need to be packaged into a single transaction to avoid any race conditions.

Although this approach is functionally correct, it's costly. There are two issues here: transactions are slow, and executing one per request would be very expensive as the data store would have to scale linearly with the number of requests. Also, because the data store is a hard dependency, the service will become unavailable if it can't reach it.

Let's address these issues. Rather than using transactions, we can use a single atomic *get-and-increment* operation that most data stores provide. Alternatively, the same can be emulated with a *compare-and-swap*. These atomic operations have much better performance than transactions.

Now, rather than updating the data store on each request, the process can batch bucket updates in memory for some time and flush them asynchronously to the data store at the end of it (see Figure 28.5). This reduces the shared state's accuracy, but it's a good trade-off as it reduces the load on the data store and the number of requests sent to it.

What happens if the data store is down? Remember the CAP theorem's essence: when there is a network fault, we can either sacrifice consistency and keep our system up or maintain consistency and stop serving requests. In our case, temporarily rejecting requests just because the data store used for rate-limiting is not reachable could damage the business. Instead, it's safer to keep serving re-

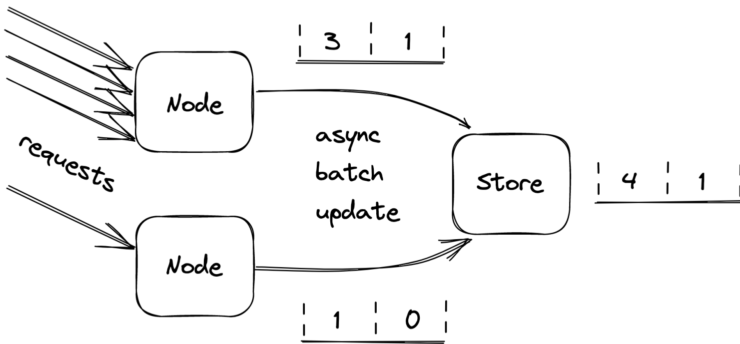

Figure 28.5: Servers batch bucket updates in memory for some time, and flush them asynchronously to the data store at the end of it.

quests based on the last state read from the store.[4]

28.4 Constant work

When overload, configuration changes, or faults force an application to behave differently from usual, we say the application has a multi-modal behavior. Some of these *modes* might trigger rare bugs, conflict with mechanisms that assume the happy path, and more generally make life harder for operators, since their mental model of how the application behaves is no longer valid. Thus, as a general rule of thumb, we should strive to minimize the number of modes.

For example, simple key-value stores are favored over relational databases in data planes because they tend to have predictable performance[5]. A relational database has many operational modes due to hidden optimizations, which can change how specific queries perform from one execution to another. Instead, dumb key-value stores behave predictably for a given query, which guarantees that

[4]This is another application of the concept of *static stability*, first introduced in chapter 22.

[5]"Some opinionated thoughts on SQL databases," https://blog.nelhage.com/post/some-opinionated-sql-takes/

there won't be any surprises.

A common reason for a system to change behavior is overload, which can cause the system to become slower and degrade at the worst possible time. Ideally, the worst- and average-case behavior shouldn't differ. One way to achieve that is by exploiting the *constant work pattern*, which keeps the work per unit time constant.

The idea is to have the system perform the same amount of work[6] under high load as under average load. And, if there is any variation under stress, it should be because the system is performing better, not worse. Such a system is also said to be *antifragile*. This is a different property from resiliency; a resilient system keeps operating under extreme load, while an antifragile one performs better.

We have already seen one application of the constant work pattern when discussing the propagation of configuration changes from the control plane to the data plane in chapter 22. For example, suppose we have a configuration store (control plane) that stores a bag of settings for each user, like the quotas used by the API gateway (data plane) to rate-limit requests. When a setting changes for a specific user, the control plane needs to broadcast it to the data plane. However, as each change is a separate independent unit of work, the data plane needs to perform work proportional to the number of changes.

If you don't see how this could be a problem, imagine that a large number of settings are updated for the majority of users at the same time (e.g., quotas changed due to a business decision). This could cause an unexpectedly large number of individual update messages to be sent to every data plane instance, which could struggle to handle them.

The workaround to this problem is simple but powerful. The control plane can periodically dump the settings of all users to a file in a scalable and highly available file store like Azure Storage or AWS S3. The dump includes the configuration settings of all users, even the ones for which there were no changes. Data plane instances

[6] "Reliability, constant work, and a good cup of coffee," https://aws.amazon.com/builders-library/reliability-and-constant-work/

can then periodically read the dump in bulk and refresh their local view of the system's configuration. Thus, no matter how many settings change, the control plane periodically writes a file to the data store, and the data plane periodically reads it.

We can take this pattern to the extreme and pre-allocate empty configuration slots for the maximum number of supported users. This guarantees that as the number of users grows, the work required to propagate changes remains stable. Additionally, doing so allows to stress-test the system and understand its behavior, knowing that it will behave the same under all circumstances. Although this limits the number of users, a limit exists regardless of whether the constant work pattern is used or not. This approach is typically used in cellular architectures (see 26.2), where a single cell has a well-defined maximum size and the system is scaled out by creating new cells.

The beauty of using the constant work pattern is that the data plane periodically performs the same amount of work in bulk, no matter how many configuration settings have changed. This makes updating settings reliable and predictable. Also, periodically writing and reading a large file is much simpler to implement correctly than a complex mechanism that only sends what changed.

Another advantage of this approach is that it's robust against a whole variety of faults thanks to its self-healing properties. If the configuration dump gets corrupted for whatever reason, no harm is done since the next update will fix it. And if a faulty update was pushed to all users by mistake, reverting it is as simple as creating a new dump and waiting it out. In contrast, the solution that sends individual updates is much harder to implement correctly, as the data plane needs complex logic to handle and heal from corrupted updates.

To sum up, performing constant work is more expensive than doing just the necessary work. Still, it's often worth considering it, given the increase in reliability and reduction in complexity it enables.

Summary

As the number of components or operations in a system increases, so does the number of failures, and eventually, anything that can happen will happen. In my team, we humorously refer to this as "cruel math."

If you talk to an engineer responsible for a production system, you will quickly realize that they tend to be more worried about minimizing and tolerating failures than how to further scale out their systems. The reason for that is that once you have an architecture that scales to handle your current load, you will be mostly concerned with tolerating faults until you eventually reach the next scalability bottleneck.

What you should take away from this part is that failures are inevitable since, no matter how hard you try, it's just impossible to build an infallible system. When you can't design away some failure modes (e.g., with constant work), the best you can do is to reduce their blast radius and stop the cracks from propagating from one component to the other.

Part V

Maintainability

Introduction

> *"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?"*
>
> – Brian Kernighan

It's a well-known fact that the majority of the cost of software is spent after its initial development in maintenance activities, such as fixing bugs, adding new features, and operating it. Thus, we should aspire to make our systems easy to modify, extend and operate so that they are easy to maintain.

Good testing — in the form of unit, integration, and end-to-end tests — is a minimum requirement to be able to modify or extend a system without worrying it will break. And once a change has been merged into the codebase, it needs to be released to production safely without affecting the application's availability. Also, the operators need to be able to monitor the system's health, investigate degradations and restore the service when it gets into a bad state. This requires altering the system's behavior without code changes, e.g., toggling a feature flag or scaling out a service with a configuration change.

Historically, developers, testers, and operators were part of different teams. First, the developers handed over their software to a team of QA engineers responsible for testing it. Then, when the software passed that stage, it moved to an operations team responsible for deploying it to production, monitoring it, and responding

to alerts. However, this model is being phased out in the industry. Nowadays, it's common for developers to be responsible for testing and operating the software they write, which requires embracing an end-to-end view of the software's lifecycle.

In this part, we will explore some of the best practices for testing and operating large distributed applications.

Chapter 29 describes the different types of tests — unit, integration, and end-to-end tests — that we can leverage to increase the confidence that a distributed application works as expected. This chapter also explores the use of formal verification methods to verify the correctness of an application before writing a single line of code.

Chapter 30 dives into continuous delivery and deployment pipelines to release changes safely and efficiently to production.

Chapter 31 discusses how to use metrics, service-level indicators, and dashboards to monitor the health of distributed applications. It then talks about how to define and enforce service-level objectives that describe how the service should behave when it's functioning correctly.

Chapter 32 introduces the concept of observability and its relation to monitoring. Later, it describes how to debug production issues using logs and traces.

Chapter 33 describes how to modify a system's behavior without changing its code, which is a must-have to enable operators to quickly mitigate failures in production when everything else fails.

Chapter 29

Testing

The longer it takes to detect a bug, the more expensive it becomes to fix. A software test can verify that some part of the application works as intended, catching bugs early in the process. But the real benefit that comes from testing shows up only later when developers want to make changes to the existing implementation (e.g., bug fixes, refactorings, and new features) without breaking the expected behaviors. Tests also act as always up-to-date documentation and improve the quality of public interfaces since developers have to put themselves in the users' shoes to test them effectively.

Unfortunately, testing is not a silver bullet because it's impossible to predict all the states a complex distributed application can get into. It only provides best-effort guarantees that the code being tested is correct and fault-tolerant. No matter how exhaustive the test coverage is, tests can only cover failures developers can imagine, not the kind of complex emergent behavior that manifests itself in production[1].

Although tests can't guarantee that code is bug-free, they certainly do a good job validating expected behaviors. So, as a rule of thumb,

[1] Cindy Sridharan wrote a great blog post series on the topic, see "Testing Microservices, the sane way," https://copyconstruct.medium.com/testing-microservices-the-sane-way-9bb31d158c16

if you want to be confident that your implementation behaves in a certain way, you have to add a test for it.

29.1 Scope

Tests come in different shapes and sizes. To begin with, we need to distinguish between the code paths a test is actually testing (aka system under test or SUT) from the ones that are being run. The SUT represents the scope of the test. It determines whether the test is categorized as a unit test, an integration test, or an end-to-end test.

A *unit test* validates the behavior of a small part of the codebase, like an individual class[2]. A good unit test should be relatively static in time and change only when the behavior of the SUT changes — refactoring, bug fixes, or new features shouldn't break it. To achieve that, a unit test should:

- use only the public interfaces of the SUT;
- test for state changes in the SUT (not predetermined sequences of actions);
- test for behaviors, i.e., how the SUT handles a given input when it's in a specific state.

An *integration test* has a larger scope than a unit test, since it verifies that a service can interact with an external dependency as expected. Confusingly, integration testing has different meanings for different people. Martin Fowler[3] makes the distinction between *narrow* and *broad* integration tests. A narrow integration test exercises only the code paths of a service that communicate with a specific external dependency, like an adapter and its supporting classes. In contrast, a broad integration test exercises code paths across multiple live services. In the rest of the chapter, we will refer to these broader integration tests as end-to-end tests.

[2]The kind of unit test I have in mind is the sociable kind; see "On the Diverse And Fantastical Shapes of Testing," https://martinfowler.com/articles/2021-test-shapes.html

[3]"IntegrationTest," https://martinfowler.com/bliki/IntegrationTest.html

An *end-to-end test* validates behavior that spans multiple services in the system, like a user-facing scenario. These tests usually run in shared environments, like staging or production, and therefore should not impact other tests or users sharing the same environment. Because of their scope, they are slow and more prone to intermittent failures.

End-to-end tests can also be painful and expensive to maintain. For example, when an end-to-end test fails, it's generally not obvious which service caused the failure, and a deeper investigation is required. But these tests are a necessary evil to ensure that user-facing scenarios work as expected across the entire application. They can uncover issues that tests with smaller scope can't, like unanticipated side effects and emergent behaviors.

One way to minimize the number of end-to-end tests is to frame them as user journey tests. A *user journey test* simulates a multi-step interaction of a user with the system (e.g., for an e-commerce service: create an order, modify it, and finally cancel it). Such a test usually requires less time to run than the individual journey parts split into separate end-to-end tests.

As the scope of a test increases, it becomes more brittle, slow, and costly. Intermittently failing tests are nearly as bad as no tests at all, as developers stop trusting them and eventually ignore their failures. When possible, it's preferable to have tests with smaller scope as they tend to be more reliable, faster, and cheaper. A good trade-off is to have a large number of unit tests, a smaller fraction of integration tests, and even fewer end-to-end tests (see Figure 29.1).

29.2 Size

The size of a test[4] reflects how much computing resources it needs to run, like the number of nodes. Generally, that depends on how realistic the environment is where the test runs. Although

[4] "Software Engineering at Google: Lessons Learned from Programming Over Time," https://www.amazon.com/dp/B0859PF5HB

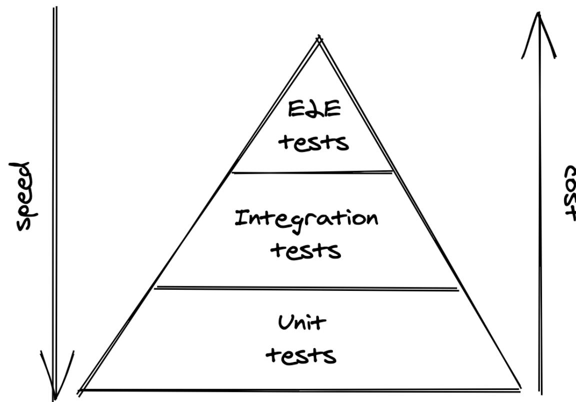

Figure 29.1: Test pyramid

the scope and size of a test tend to be correlated, they are distinct concepts, and it helps to separate them.

A *small test* runs in a single process and doesn't perform any blocking calls or I/O. As a result, it's very fast, deterministic, and has a very small probability of failing intermittently.

An *intermediate test* runs on a single node and performs local I/O, like reads from disk or network calls to localhost. This introduces more room for delays and non-determinism, increasing the likelihood of intermittent failures.

A *large test* requires multiple nodes to run, introducing even more non-determinism and longer delays.

Unsurprisingly, the larger a test is, the longer it takes to run and the flakier it becomes. This is why we should write the smallest possible test for a given behavior. We can use a *test double* in place of a real dependency, such as a fake, a stub, or a mock, to reduce the test's size, making it faster and less prone to intermittent failures:

- A *fake* is a lightweight implementation of an interface that behaves similarly to a real one. For example, an in-memory version of a database is a fake.
- A *stub* is a function that always returns the same value no

matter which arguments are passed to it.
- Finally, a *mock* has expectations on how it should be called, and it's used to test the interactions between objects.

The problem with test doubles is that they don't resemble how the real implementation behaves with all its nuances. The weaker the resemblance is, the less confidence we should have that the test using the double is actually useful. Therefore, when the real implementation is fast, deterministic, and has few dependencies, we should use that rather than a double. When using the real implementation is not an option, we can use a fake maintained by the same developers of the dependency if one is available. Stubbing, or mocking, are last-resort options as they offer the least resemblance to the actual implementation, which makes tests that use them brittle.

For integration tests, a good compromise is to combine mocking with contract tests[5]. A *contract test* defines a request for an external dependency with the corresponding expected response. Then the test uses this contract to mock the dependency. For example, a contract for a REST API consists of an HTTP request and response. To ensure that the contract is valid and doesn't break in the future, the test suite of the external dependency uses the same contract definition to simulate the client request and ensure that the expected response is returned.

29.3 Practical considerations

As with everything else, testing requires making trade-offs. Suppose we want to end-to-end test the behavior of a specific API endpoint exposed by a service. The service talks to:

- a data store,
- an internal service owned by another team,
- and a third-party API used for billing (see Figure 29.2).

As suggested earlier, we should try to write the smallest possible

[5]"ContractTest," https://martinfowler.com/bliki/ContractTest.html

test for the desired scope while minimizing the use of test doubles that don't resemble how the real implementation behaves.

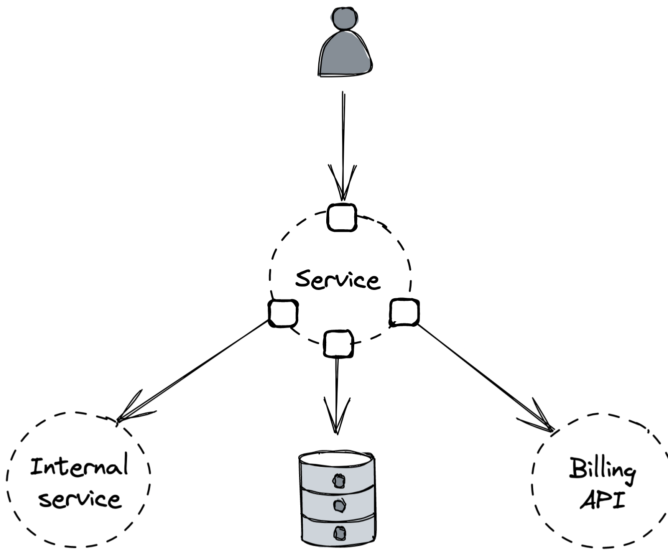

Figure 29.2: How should we test the service?

Let's assume the specific endpoint under test doesn't communicate with the internal service, so we can safely use a mock in its place. And if the data store comes with an in-memory implementation (a fake), we can use that in the test to avoid issuing network calls. Finally, we can't easily call the third-party billing API since that would require issuing real transactions. However, assuming a fake is not available, the billing service might still offer a testing endpoint that issues fake transactions.

Here is a more nuanced example in which it's a lot riskier to go for a smaller test. Suppose we need to test whether purging the data belonging to a specific user across the entire application stack works as expected. In Europe, this functionality is mandated by law (GDPR), and failing to comply with it can result in fines up to 20 million euros or 4% annual turnover, whichever is greater. In this case, because the risk of the functionality silently breaking is high, we want to be as confident as possible that it's working as

expected. This warrants the use of an end-to-end test that runs in production periodically and uses live services rather than test doubles.

29.4 Formal verification

Software tests are not the only way to catch bugs early. Taking the time to write a high-level description of how a system behaves, i.e., a *specification*, allows subtle bugs and architecture shortcomings to be detected before writing a single line of code.

A specification can range from an informal one-pager to a formal mathematical description that a computer can check. Since it's hard to specify what we don't fully understand, a specification can help us reason about the behaviors of the system we are designing. It also acts as documentation for others and as a guide for the actual implementation. On top of the benefits mentioned so far, by writing the specification in a formal language, we also gain the ability to verify algorithmically whether the specification is flawed (*model checking*).

Writing a specification doesn't mean describing every corner of a system in detail. The specification's goal is to catch errors while they are still cheap to fix. Therefore, we only want to specify those parts that are most likely to contain errors and are hard to detect by other means, like traditional tests. Once we have decided what to specify, we also need to choose the level of abstraction, i.e., which details to omit.

TLA+[6] is a well-known and widely used formal specification language[7]. The likes of Amazon or Microsoft use it to describe some of their most complex distributed systems, like S3 or Cosmos DB[8].

In TLA+, a *behavior* of a system is represented by a sequence of states, where a state is an assignment of values to global variables.

[6] "The TLA+ Home Page," https://lamport.azurewebsites.net/tla/tla.html

[7] TLA stands for Temporal Logic of Actions.

[8] "Industrial Use of TLA+," https://lamport.azurewebsites.net/tla/industrial-use.html

Thus, the specification of a system is the set of all possible behaviors.

One of the goals of writing a specification is to verify that it satisfies properties we want the system to have, like safety and liveness. A *safety* property asserts that something is true for all states of a behavior (invariant). A *liveness* property instead asserts that something eventually happens. TLA+ allows to describe and verify properties that should be satisfied by all possible states and behaviors of a specification. This is extremely powerful, since a system running at scale will eventually run into all possible states and behaviors, and humans are bad at imagining behaviors in which several rare events occur simultaneously.

For example, suppose we have a service that uses key-value store X, and we would like to migrate it to use key-value store Y that costs less and has proven to perform better in benchmarks. At a high level, one way we could implement this migration without any downtime is the following:

1. The service writes to both data stores X and Y (dual write) while reading exclusively from X.
2. A one-off batch process backfills Y with data from X created before the service started writing to Y.
3. The application switches to read and write exclusively from and to Y.

This approach might seem reasonable, but will it guarantee that the data stores eventually end up in the same state?

If we were to model this with TLA+, the model checker would be able to identify several problems, like a liveness violation that leaves the system in an inconsistent state when a service instance crashes after writing to A but before writing to B. The beauty of automated model checking is that it returns an error trace with the behavior (i.e., sequence of states) that violates the properties when it fails.

Although modeling writes as atomic (i.e., either both writes succeed, or they both fail) fixes the liveness issue, the model isn't cor-

rect yet. For example, if two service instances are writing to A and B simultaneously, the two data stores can end up in different states because the order of writes can differ, as shown in Fig 29.3.

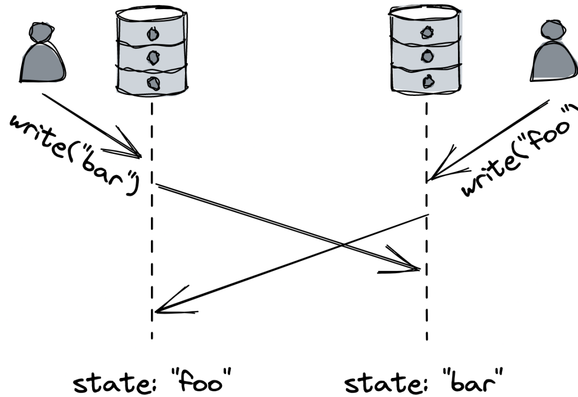

Figure 29.3: Data stores can see writes in different orders.

If you remember the discussion about transactions in section 13.1, you know we can solve this problem by introducing a message channel between the service and data stores that serializes all writes and guarantees a single global order. Regardless of the actual solution, the point is that a formal model enables us to test architectural decisions that would be hard to verify otherwise.

Chapter 30

Continuous delivery and deployment

Once a change and its newly introduced tests have been merged to a repository, it needs to be released to production. When releasing a change requires a manual process, it won't happen frequently. This means that several changes, possibly over days or even weeks, end up being batched and released together, increasing the likelihood of the release failing. And when a release fails, it's harder to pinpoint the breaking change[1], slowing down the team. Also, the developer who initiated the release needs to keep an eye on it by monitoring dashboards and alerts to ensure that it's working as expected or roll it back.

Manual deployments are a terrible use of engineering time. The problem gets further exacerbated when there are many services. Eventually, the only way to release changes safely and efficiently is to automate the entire process. Once a change has been merged to a repository, it should automatically be rolled out to production safely. The developer is then free to context-switch to their next task rather than shepherding the deployment. The whole release process, including rollbacks, can be automated with a continuous

[1] There could be multiple breaking changes, actually.

delivery and deployment (CD) pipeline.

Because releasing changes is one of the main sources of failures, CD requires a significant amount of investment in terms of safeguards, monitoring, and automation. If a regression is detected, the artifact being released — i.e., the deployable component that includes the change — is either rolled back to the previous version or forward to the next one, assuming it contains a hotfix.

There is a balance between the safety of a rollout and the time it takes to release a change to production. A good CD pipeline should strive to make a good trade-off between the two. In this chapter, we will explore how.

30.1 Review and build

At a high level, a code change needs to go through a pipeline of four stages to be released to production: review, build, pre-production rollout, and production rollout.

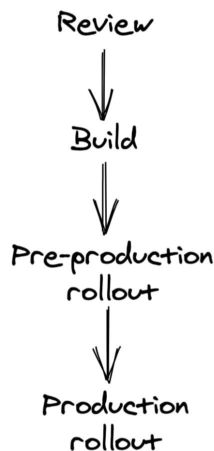

Figure 30.1: Continuous delivery and deployment pipeline stages

It all starts with a pull request (PR) submitted for review by a developer to a repository. When the PR is submitted for review, it needs

to be compiled, statically analyzed, and validated with a battery of tests, all of which shouldn't take longer than a few minutes. To increase the tests' speed and minimize intermittent failures, the tests that run at this stage should be small enough to run on a single process or node, while larger tests run later in the pipeline.

The PR needs to be reviewed and approved by a team member before it can be merged into the repository. The reviewer has to validate whether the change is correct and safe to be released to production automatically by the CD pipeline. A checklist can help the reviewer not to forget anything important, e.g.:

- Does the change include unit, integration, and end-to-end tests as needed?
- Does the change include metrics, logs, and traces?
- Can this change break production by introducing a backward-incompatible change or hitting some service limit?
- Can the change be rolled back safely, if needed?

Code changes shouldn't be the only ones going through this review process. For example, static assets, end-to-end tests, and configuration files should all be version-controlled in a repository (not necessarily the same one) and be treated just like code. The same service can then have multiple CD pipelines, one for each repository, potentially running in parallel.

I can't stress enough the importance of reviewing and releasing configuration changes with a CD pipeline. As discussed in chapter 24, one of the most common causes of production failures are configuration changes applied globally without any prior review or testing.

Also, applications running in the cloud should declare their infrastructure dependencies, like virtual machines, data stores, and load balancers, with code (aka *Infrastructure as Code (IaC)*[2]) using tools

[2] "What is Infrastructure as Code?," https://docs.microsoft.com/en-us/devops/deliver/what-is-infrastructure-as-code

like Terraform[3]. This allows the provisioning of infrastructure to be automated and infrastructure changes to be treated just like any other software change.

Once a change has been merged into its repository's main branch, the CD pipeline moves to the build stage, in which the repository's content is built and packaged into a deployable release artifact.

30.2 Pre-production

During this stage, the artifact is deployed and released to a synthetic pre-production environment. Although this environment lacks the realism of production, it's useful to verify that no hard failures are triggered (e.g., a null pointer exception at startup due to a missing configuration setting) and that end-to-end tests succeed. Because releasing a new version to pre-production requires significantly less time than releasing it to production, bugs can be detected earlier.

There can be multiple pre-production environments, starting with one created from scratch for each artifact and used to run simple smoke tests, to a persistent one similar to production that receives a small fraction of mirrored requests from it. AWS, for example, is known for using multiple pre-production environments[4].

Ideally, the CD pipeline should assess the artifact's health in pre-production using the same health signals used in production. Metrics, alerts, and tests used in pre-production should be equivalent to those used in production, to avoid the former becoming a second-class citizen with sub-par health coverage.

[3] "Terraform: an open-source infrastructure as code software tool," https://www.terraform.io/

[4] "Automating safe, hands-off deployments," https://aws.amazon.com/builders-library/automating-safe-hands-off-deployments/

30.3 Production

Once an artifact has been rolled out to pre-production successfully, the CD pipeline can proceed to the final stage and release the artifact to production. It should start by releasing it to a small number of production instances at first[5]. The goal is to surface problems that haven't been detected so far as quickly as possible before they have the chance to cause widespread damage in production.

If that goes well and all the health checks pass, the artifact is incrementally released to the rest of the fleet. While the rollout is in progress, a fraction of the fleet can't serve any traffic due to the ongoing deployment, and so the remaining instances need to pick up the slack. For this to not cause any performance degradation, there needs to be enough capacity left to sustain the incremental release.

If the service is available in multiple regions, the CD pipeline should first start with a low-traffic region to reduce the impact of a faulty release. Then, releasing the remaining regions should be divided into sequential stages to minimize risks further. Naturally, the more stages there are, the longer the CD pipeline takes to release the artifact to production. One way to mitigate this problem is by increasing the release speed once the early stages complete successfully and enough confidence has been built up. For example, the first stage could release the artifact to a single region, the second to a larger region, and the third to N regions simultaneously.

30.4 Rollbacks

After each step, the CD pipeline needs to assess whether the artifact deployed is healthy and, if not, stop the release and roll it back. A variety of health signals can be used to make that decision, such as the result of end-to-end tests, health metrics like latencies and errors and alerts.

[5] This is also referred to as canary testing.

Monitoring just the health signals of the service being rolled out is not enough. The CD pipeline should also monitor the health of upstream and downstream services to detect any indirect impact of the rollout. The pipeline should allow enough time to pass between one step and the next (bake time) to ensure that it was successful, as some issues only appear after some time has passed. For example, a performance degradation could be visible only at peak time. To speed up the release, the bake time can be reduced after each step succeeds and confidence is built up. The CD pipeline could also gate the bake time on the number of requests seen for specific API endpoints to guarantee that the API surface has been properly exercised.

When a health signal reports a degradation, the CD pipeline stops. At that point, it can either roll back the artifact automatically or trigger an alert to engage the engineer on call, who needs to decide whether a rollback is warranted or not[6]. Based on the engineer's input, the CD pipeline retries the stage that failed (e.g., perhaps because something else was going into production at the time) or rolls back the release entirely.

The operator can also stop the pipeline and wait for a new artifact with a hotfix to be rolled forward. This might be necessary if the release can't be rolled back because of a backward-incompatible change. Since rolling forward is much riskier than rolling back, any change introduced should always be backward compatible as a rule of thumb. One of the most common causes for backward incompatibility is changing the serialization format used for persistence or IPC purposes.

To safely introduce a backward-incompatible change, it needs to be broken down into multiple backward-compatible changes[7]. For example, suppose the messaging schema between a producer and a consumer service needs to change in a backward-incompatible way. In this case, the change is broken down into three smaller

[6] CD pipelines can be configured to run only during business hours to minimize the disruption to on-call engineers.

[7] "Ensuring rollback safety during deployments," https://aws.amazon.com/builders-library/ensuring-rollback-safety-during-deployments/

changes that can individually be rolled back safely:

- In the *prepare change*, the consumer is modified to support both the new and old messaging format.
- In the *activate change*, the producer is modified to write the messages in the new format.
- Finally, in the *cleanup change*, the consumer stops supporting the old messaging format altogether. This change is only released once there is enough confidence that the activated change won't need to be rolled back.

An automated upgrade-downgrade test part of the CD pipeline in pre-production can be used to validate whether a change is actually safe to roll back.

Chapter 31

Monitoring

Monitoring is primarily used to detect failures that impact users in production and to trigger notifications (or alerts) to the human operators responsible for the system. Another important use case for monitoring is to provide a high-level overview of the system's health via dashboards.

In the early days, monitoring was used mostly to report whether a service was up, without much visibility of what was going on inside (black-box monitoring). In time, developers also started to instrument their applications to report whether specific features worked as expected (white-box monitoring). This was popularized with the introduction of statsd[1] by Etsy, which normalized collecting application-level measurements. While black-box monitoring is useful for detecting the symptoms of a failure, white-box monitoring can help identify the root cause.

The main use case for black-box monitoring is to monitor external dependencies, such as third-party APIs, and validate how users perceive the performance and health of a service from the outside. A common black-box approach is to periodically run scripts (*syn-*

[1] "Measure Anything, Measure Everything," https://codeascraft.com/2011/02/15/measure-anything-measure-everything/

thetics[2]) that send test requests to external API endpoints and monitor how long they took and whether they were successful. Synthetics are deployed in the same regions the application's users are and hit the same endpoints they do. Because they exercise the system's public surface from the outside, they can catch issues that aren't visible from within the application, like connectivity problems. Synthetics are also useful for detecting issues with APIs that aren't exercised often by users.

For example, if the DNS server of a service were down, the issue would be visible to synthetics, since they wouldn't be able to resolve its IP address. However, the service itself would think everything was fine, and it was just getting fewer requests than usual.

31.1 Metrics

A *metric* is a time series of raw measurements (samples) of resource usage (e.g., CPU utilization) or behavior (e.g., number of requests that failed), where each sample is represented by a floating-point number and a timestamp.

Commonly, a metric can also be tagged with a set of key-value pairs (*labels*). For example, the label could represent the region, data center, cluster, or node where the service is running. Labels make it easy to slice and dice the data and eliminate the instrumentation cost of manually creating a metric for each label combination. However, because every distinct combination of labels is a different metric, tagging generates a large number of metrics, making them challenging to store and process.

At the very least, a service should emit metrics about its load (e.g., request throughput), its internal state (e.g., in-memory cache size), and its dependencies' availability and performance (e.g., data store response time). Combined with the metrics emitted by downstream services, this allows operators to identify problems quickly. But this requires explicit code changes and a deliberate

[2] "Using synthetic monitoring," https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/CloudWatch_Synthetics_Canaries.html

effort by developers to instrument their code.

For example, take a fictitious HTTP handler that returns a resource. There is a whole range of questions we will want to be able to answer once it's running in production[3]:

```
def get_resource(id):
  resource = self._cache.get(id)  # in-process cache
  # Is the id valid?
  # Was there a cache hit?
  # How long has the resource been in the cache?

  if resource is not None:
    return resource

  resource = self._repository.get(id)
  # Did the remote call fail, and if so, why?
  # Did the remote call time out?
  # How long did the call take?

  self._cache[id] = resource
  # What's the size of the cache?

  return resource
  # How long did it take for the handler to run?
```

Now, suppose we want to record the number of requests the handler failed to serve. One way to do that is with an event-based approach — whenever the handler fails to handle a request, it reports a failure count of 1 in an *event*[4] to a local telemetry agent, e.g.:

```
{
  "failureCount": 1,
  "serviceRegion": "EastUs2",
  "timestamp": 1614438079
```

[3] I have omitted error handling for simplicity.

[4] We will talk more about event logs in section 32.1; for now, assume an event is just a dictionary.

```
}
```

The agent batches these events and emits them periodically to a remote telemetry service, which persists them in a dedicated data store for event logs. For example, this is the approach taken by Azure Monitor's log-based metrics[5].

As you can imagine, this is quite expensive, since the load on the telemetry service increases with the number of events ingested. Events are also costly to aggregate at query time — suppose we want to retrieve the number of failures in North Europe over the past month; we would have to issue a query that requires fetching, filtering, and aggregating potentially trillions of events within that time period.

So is there a way to reduce costs at query time? Because metrics are time series, they can be modeled and manipulated with mathematical tools. For example, time-series samples can be pre-aggregated over fixed time periods (e.g., 1 minute, 5 minutes, 1 hour, etc.) and represented with summary statistics such as the sum, average, or percentiles.

Going back to our example, the telemetry service could pre-aggregate the failure count events at ingestion time. If the aggregation (i.e., the sum in our example) were to happen with a period of one hour, we would have one *failureCount* metric per *serviceRegion*, each containing one sample per hour, e.g.:

```
"00:00", 561,
"01:00", 42,
"02:00", 61,
...
```

The ingestion service could also create multiple pre-aggregates with different periods. This way, the pre-aggregated metric with the best period that satisfies the query can be chosen at query

[5] "Log-based metrics in Application Insights," https://docs.microsoft.com/en-us/azure/azure-monitor/app/pre-aggregated-metrics-log-metrics#log-based-metrics

time. For example, CloudWatch[6] (the telemetry service used by AWS) pre-aggregates data as it's ingested.

We can take this idea one step further and also reduce ingestion costs by having the local telemetry agents pre-aggregate metrics client-side. By combining client- and server-side pre-aggregation, we can drastically reduce the bandwidth, compute, and storage requirements for metrics. However, this comes at a cost: we lose the ability to re-aggregate metrics after ingestion because we no longer have access to the original events that generated them. For example, if a metric is pre-aggregated over a period of 1 hour, it can't later be re-aggregated over a period of 5 minutes without the original events.

Because metrics are mainly used for alerting and visualization purposes, they are usually persisted in a pre-aggregated form in a data store specialized for efficient time series storage[7].

31.2 Service-level indicators

As noted before, one of the main use cases for metrics is alerting. But that doesn't mean we should create alerts for every possible metric — for example, it's useless to be alerted in the middle of the night because a service had a big spike in memory consumption a few minutes earlier.

In this section, we will discuss one specific metric category that lends itself well to alerting: *service-level indicators* (SLIs). An SLI is a metric that measures one aspect of the *level of service* provided by a service to its users, like the response time, error rate, or throughput. SLIs are typically aggregated over a rolling time window and represented with a summary statistic, like an average or percentile.

SLIs are best defined as a ratio of two metrics: the number of "good events" over the total number of events. That makes the ratio easy

[6] "Amazon CloudWatch concepts," https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/cloudwatch_concepts.html

[7] like, e.g., Druid; see "A Real-time Analytical Data Store," http://static.druid.io/docs/druid.pdf

to interpret: 0 means the service is completely broken and 1 that whatever is being measured is working as expected (see Figure 31.1). As we will see later in the chapter, ratios also simplify the configuration of alerts. Some commonly used SLIs for services are:

- *Response time* — The fraction of requests that are completed faster than a given threshold.
- *Availability* — The proportion of time the service was usable, defined as the number of successful requests over the total number of requests.

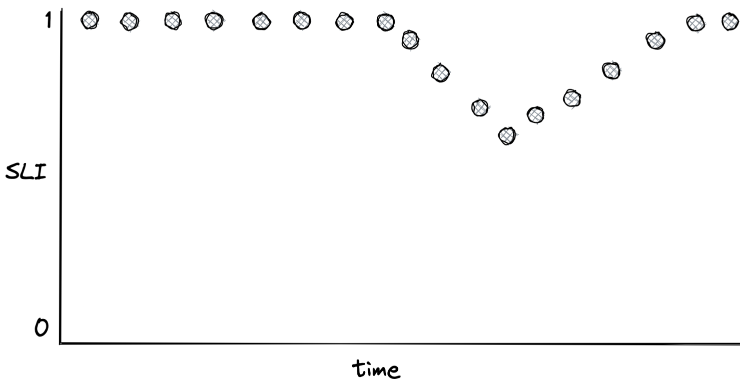

Figure 31.1: An SLI defined as the ratio of good events over the total number of events

Once we have decided what to measure, we need to decide where to measure it. Take the response time, for example. Should we measure the response time as seen by the service, load balancer, or clients? Ideally, we should select the metric that best represents the users' experience. If that's too costly to collect, we should pick the next best candidate. In the previous example, the client metric is the most meaningful of the lot since it accounts for delays and hiccups through the entire request path.

Now, how should we measure the response time? Measurements can be affected by many factors, such as network delays, page faults, or heavy context switching. Since every request does not take the same amount of time, response times are best repre-

sented with a distribution, which usually is right-skewed and long-tailed[8].

A distribution can be summarized with a statistic. Take the average, for example. While it has its uses, it doesn't tell us much about the proportion of requests experiencing a specific response time, and all it takes to skew the average is one large outlier. For example, suppose we collected 100 response times, of which 99 are 1 second, and one is 10 minutes. In this case, the average is nearly 7 seconds. So even though 99% of the requests experience a response time of 1 second, the average is 7 times higher than that.

A better way to represent the distribution of response times is with percentiles. A percentile is the value below which a percentage of the response times fall. For example, if the 99th percentile is 1 second, then 99% of requests have a response time below or equal to 1 second. The upper percentiles of a response time distribution, like the 99th and 99.9th percentiles, are also called *long-tail latencies*. Even though only a small fraction of requests experience these extreme latencies, it can impact the most important users for the business. They are the ones that make the highest number of requests and thus have a higher chance of experiencing tail latencies. There are studies[9] that show that high latencies negatively affect revenues: a mere 100-millisecond delay in load time can hurt conversion rates by 7 percent.

Also, long-tail latencies can dramatically impact a service. For example, suppose a service on average uses about 2K threads to serve 10K requests per second. By Little's Law[10], the average response time of a thread is 200 ms. Now, if suddenly 1% of requests start taking 20 seconds to complete (e.g., because of a congested switch and relaxed timeouts), 2K additional threads are needed to deal

[8] "latency: a primer," https://igor.io/latency/

[9] "Akamai Online Retail Performance Report: Milliseconds Are Critical," https://www.akamai.com/newsroom/press-release/akamai-releases-spring-2017-state-of-online-retail-performance-report

[10] Little's law says the average number of items in a system equals the average rate at which new items arrive multiplied by the average time an item spends in the system; see "Back of the envelope estimation hacks," https://robertovitillo.com/back-of-the-envelope-estimation-hacks/

just with the slow requests. So the number of threads used by the service has to double to sustain the load!

Measuring long-tail latencies and keeping them in check doesn't just make our users happy but also drastically improves the resiliency of our systems while reducing their operational costs. Intuitively, by reducing the long-tail latency (worst-case scenario), we also happen to improve the average-case scenario.

31.3 Service-level objectives

A *service-level objective* (SLO) defines a range of acceptable values for an SLI within which the service is considered to be in a healthy state (see Figure 31.2). An SLO sets the expectation to the service's users of how it should behave when it's functioning correctly. Service owners can also use SLOs to define a service-level agreement (SLA) with their users — a contractual agreement that dictates what happens when an SLO isn't met, generally resulting in financial consequences.

For example, an SLO could define that 99% of API calls to endpoint X should complete below 200 ms, as measured over a rolling window of 1 week. Another way to look at it is that it's acceptable for up to 1% of requests within a rolling week to have a latency higher than 200 ms. That 1% is also called the *error budget*, which represents the number of failures that can be tolerated.

SLOs are helpful for alerting purposes and also help the team prioritize repair tasks. For example, the team could agree that when an error budget is exhausted, repair items will take precedence over new features until the SLO is repaired. Furthermore, an incident's importance can be measured by how much of the error budget has been burned. For example, an incident that burned 20% of the error budget is more important than one that burned only 1%.

Smaller time windows force the team to act quicker and prioritize bug fixes and repair items, while longer windows are better suited to make long-term decisions about which projects to invest

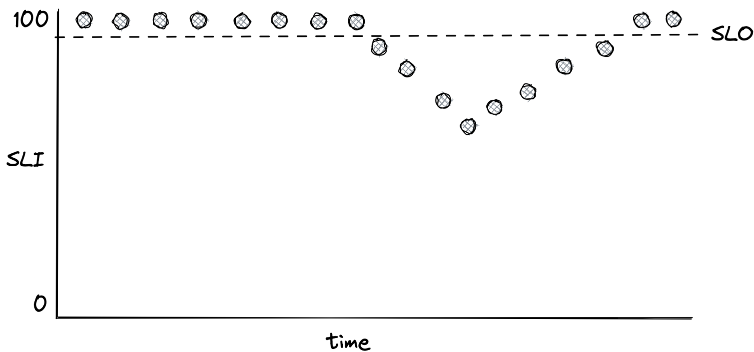

Figure 31.2: An SLO defines the range of acceptable values for an SLI.

in. Consequently, it makes sense to have multiple SLOs with different window sizes.

How strict should SLOs be? Choosing the right target range is harder than it looks. If it's too lenient, we won't detect user-facing issues; if it's too strict, engineering time will be wasted with micro-optimizations that yield diminishing returns. Even if we could guarantee 100% reliability for our systems (which is impossible), we can't make guarantees for anything that our users depend on to access our service and which is outside our control, like their last-mile connection. Thus, 100% reliability doesn't translate into a 100% reliable experience for users.

When setting the target range for SLOs, it's reasonable to start with comfortable ranges and tighten them as we build confidence. We shouldn't just pick targets our service meets today that might become unattainable in a year after load increases. Instead, we should work backward from what users care about. In general, anything above 3 nines of availability is very costly to achieve and provides diminishing returns.

Also, we should strive to keep things simple and have as few SLOs as possible that provide a good enough indication of the desired service level and review them periodically. For example, suppose we discover that a specific user-facing issue generated many sup-

port tickets, but none of our SLOs showed any degradation. In that case, the SLOs might be too relaxed or simply not capture a specific use case.

SLOs need to be agreed on with multiple stakeholders. If the error budget is being burned too rapidly or has been exhausted, repair items have to take priority over features. Engineers need to agree that the targets are achievable without excessive toil. Product managers also have to agree that the targets guarantee a good user experience. As Google's SRE book mentions[11]: "if you can't ever win a conversation about priorities by quoting a particular SLO, it's probably not worth having that SLO."

It's worth mentioning that users can become over-reliant on the actual behavior of our service rather than its documented SLA. To prevent that, we can periodically inject controlled failures[12] in production — also known as chaos testing. These controlled failures ensure the dependencies can cope with the targeted service level and are not making unrealistic assumptions. As an added benefit, they also help validate that resiliency mechanisms work as expected.

31.4 Alerts

Alerting is the part of a monitoring system that triggers an action when a specific condition happens, like a metric crossing a threshold. Depending on the severity and the type of the alert, the action can range from running some automation, like restarting a service instance, to ringing the phone of a human operator who is on call. In the rest of this section, we will mostly focus on the latter case.

For an alert to be useful, it has to be actionable. The operator shouldn't spend time exploring dashboards to assess the alert's impact and urgency. For example, an alert signaling a spike in CPU usage is not useful as it's not clear whether it has any impact

[11] "Service Level Objectives," https://sre.google/sre-book/service-level-objectives/

[12] "Chaos engineering," https://en.wikipedia.org/wiki/Chaos_engineering

on the system without further investigation. On the other hand, an SLO is a good candidate for an alert because it quantifies the impact on the users. The SLO's error budget can be monitored to trigger an alert whenever a large fraction of it has been consumed.

Before discussing how to define an alert, it's important to understand that there is a trade-off between its precision and recall. Formally, *precision* is the fraction of significant events (i.e., actual issues) over the total number of alerts, while *recall* is the ratio of significant events that triggered an alert. Alerts with low precision are noisy and often not actionable, while alerts with low recall don't always trigger during an outage. Although it would be nice to have 100% precision and recall, improving one typically lowers the other, and so a compromise needs to be made.

Suppose we have an availability SLO of 99% over 30 days, and we would like to configure an alert for it. A naive way would be to trigger an alert whenever the availability goes below 99% within a relatively short time window, like an hour. But how much of the error budget has actually been burned by the time the alert triggers? Because the time window of the alert is one hour, and the SLO error budget is defined over 30 days, the percentage of error budget that has been spent when the alert triggers is $\frac{1 \text{ hour}}{30 \text{ days}} = 0.14\%$. A system is never 100% healthy, since there is always something failing at any given time. So being notified that 0.14% of the SLO's error budget has been burned is not useful. In this case, we have a high recall but a low precision.

We can improve the alert's precision by increasing the amount of time the condition needs to be true. The problem is that now the alert will take longer to trigger, even during an actual outage. The alternative is to alert based on how fast the error budget is burning, also known as the *burn rate*, which lowers the detection time. The burn rate is defined as the percentage of the error budget consumed over the percentage of the SLO time window that has elapsed — it's the rate of exhaustion of the error budget. So using our previous example, a burn rate of 1 means the error budget will be exhausted precisely in 30 days; if the rate is 2, then it will be 15 days; if the rate is 3, it will be 10 days, and so on.

To improve recall, we can have multiple alerts with different thresholds. For example, a burn rate below 2 could be classified as a low-severity alert to be investigated during working hours, while a burn rate over 10 could trigger an automated call to an engineer. The SRE workbook has some great examples[13] of how to configure alerts based on burn rates.

While the majority of alerts should be based on SLOs, some should trigger for known failure modes that we haven't had the time to design or debug away. For example, suppose we know a service suffers from a memory leak that has led to an incident in the past, but we haven't yet managed to track down the root cause. In this case, as a temporary mitigation, we could define an alert that triggers an automated restart when a service instance is running out of memory.

31.5 Dashboards

After alerting, the other main use case for metrics is to power real-time dashboards that display the overall health of a system. Unfortunately, dashboards can easily become a dumping ground for charts that end up being forgotten, have questionable usefulness, or are just plain confusing. Good dashboards don't happen by coincidence. In this section, we will discuss some of the best practices for creating useful dashboards.

When creating a dashboard, the first decision we have to make is to decide who the audience is[14] and what they are looking for. Then, given the audience, we can work backward to decide which charts, and therefore metrics, to include.

The categories of dashboards presented here (see Figure 31.3) are by no means standard but should give you an idea of how to organize dashboards.

SLO dashboard

[13] "Alerting on SLOs," https://sre.google/workbook/alerting-on-slos/

[14] "Building dashboards for operational visibility," https://aws.amazon.com/builders-library/building-dashboards-for-operational-visibility

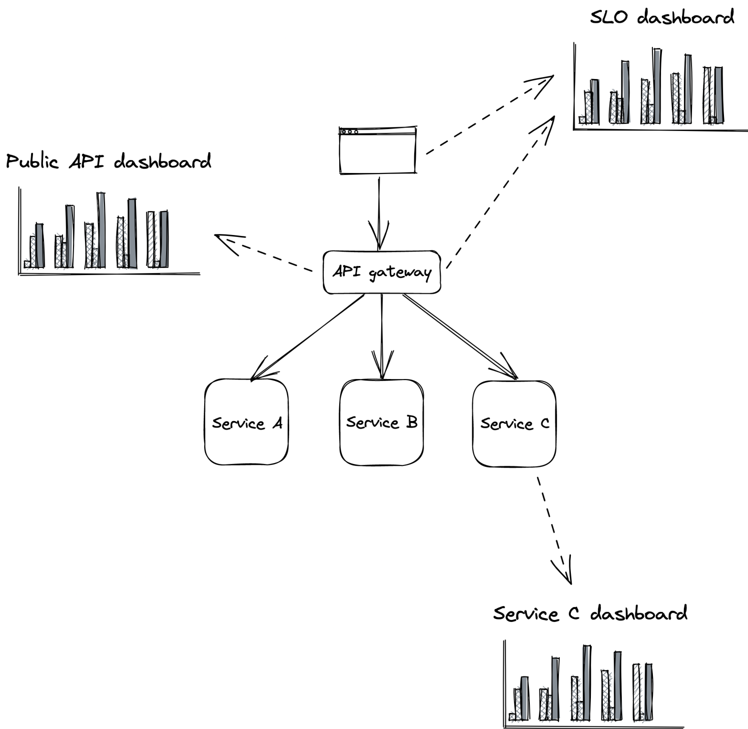

Figure 31.3: Dashboards should be tailored to their audience.

The SLO summary dashboard is designed to be used by various stakeholders from across the organization to gain visibility into the system's health as represented by its SLOs. During an incident, this dashboard quantifies the impact the incident is having on the users.

Public API dashboard

This dashboard displays metrics about the system's public API endpoints, which helps operators identify problematic paths during an incident. For each endpoint, the dashboard exposes several metrics related to request messages, request handling, and response messages, like:

- Number of requests received or messages pulled from a mes-

saging broker, request size statistics, authentication issues, etc.
- Request handling duration, availability and response time of external dependencies, etc.
- Counts per response type, size of responses, etc.

Service dashboard

A service dashboard displays service-specific implementation details, which require an in-depth understanding of its inner workings. Unlike the previous dashboards, this one is primarily used by the team that owns the service. Beyond service-specific metrics, a service dashboard should also contain metrics of upstream dependencies like load balancers and messaging queues and downstream dependencies like data stores.

This dashboard offers a first entry point into what's going on within the service when debugging. As we will later learn when discussing observability, this high-level view is just the starting point. The operator typically drills down into the metrics by segmenting them further and eventually reaches for raw logs and traces to get more detail.

31.5.1 Best practices

As new metrics are added and old ones removed, charts and dashboards need to be modified and kept in sync across multiple environments (e.g., pre-production and production). The most effective way to achieve that is by defining dashboards and charts with a domain-specific language and version-controlling them just like code. This allows dashboards to be updated from the same pull request that contains related code changes without manually updating dashboards, which is error-prone.

As dashboards render top to bottom, the most important charts should always be located at the very top. Also, charts should be rendered with a default timezone, like UTC, to ease the communication between people located in different parts of the world when looking at the same data.

All charts in the same dashboard should use the same time resolution (e.g., 1 minute, 5 minutes, 1 hour, etc.) and range (24 hours, 7 days, etc.). This makes it easy to visually correlate anomalies across charts in the same dashboard. We can pick the default time range and resolution based on the dashboard's most common use case. For example, a 1-hour range with a 1-minute resolution is best for monitoring an ongoing incident, while a 1-year range with a 1-day resolution is best for capacity planning.

We should keep the number of data points and metrics on the same chart to a minimum. Rendering too many points doesn't just make charts slow to download / render but also makes it hard to interpret them and spot anomalies.

A chart should contain only metrics with similar ranges (min and max values); otherwise, the metric with the largest range can completely hide the others with smaller ranges. For that reason, it makes sense to split related statistics for the same metric into multiple charts. For example, the 10th percentile, average and 90th percentile of a metric can be displayed in one chart, and the 0.1th percentile, 99.9th percentile, minimum and maximum in another.

A chart should also contain useful annotations, like:

- a description of the chart with links to runbooks, related dashboards, and escalation contacts;
- a horizontal line for each configured alert threshold, if any;
- a vertical line for each relevant deployment.

Metrics that are only emitted when an error condition occurs can be hard to interpret as charts will show wide gaps between the data points, leaving the operator wondering whether the service stopped emitting that metric due to a bug. To avoid this, it's best practice to emit a metric using a value of zero in the absence of an error and a value of 1 in the presence of it.

31.6 Being on call

A healthy on-call rotation is only possible when services are built from the ground up with reliability and operability in mind. By making the developers responsible for operating what they build, they are incentivized to reduce the operational toll to a minimum. They are also in the best position to be on call since they are intimately familiar with the system's architecture, brick walls, and trade-offs.

Being on call can be very stressful. Even when there are no call-outs, just the thought of not having the usual freedom outside of regular working hours can cause a great deal of anxiety. This is why being on call should be compensated, and there shouldn't be any expectations for the on-call engineer to make any progress on feature work. Since they will be interrupted by alerts, they should make the most of it and be given free rein to improve the on-call experience by, e.g., revising dashboards or improving resiliency mechanisms.

Achieving a healthy on-call rotation is only possible when alerts are actionable. When an alert triggers, to the very least, it should link to relevant dashboards and a run-book that lists the actions the engineer should take[15]. Unless the alert was a false positive, all actions taken by the operator should be communicated into a shared channel like a global chat accessible by other teams. This allows other engineers to chime in, track the incident's progress, and more easily hand over an ongoing incident to someone else.

The first step to address an alert is to mitigate it, not fix the underlying root cause that created it. A new artifact has been rolled out that degrades the service? Roll it back. The service can't cope with the load even though it hasn't increased? Scale it out.

Once the incident has been mitigated, the next step is to understand the root cause and come up with ways to prevent it from happening again. The greater the impact was, as measured by

[15] Ideally, we should automate what we can to minimize manual actions that operators need to perform. As it turns out, machines are good at following instructions.

the SLOs, the more time we should spend on this. Incidents that burned a significant fraction of an SLO's error budget require a formal *postmortem*. The goal of the postmortem is to understand the incident's root cause and come up with a set of repair items that will prevent it from happening again. Ideally, there should also be an agreement in the team that if an SLO's error budget is burned or the number of alerts spirals out of control, the whole team stops working on new features to focus exclusively on reliability until a healthy on-call rotation has been restored.

The SRE books[16] provide a wealth of information and best practices regarding setting up a healthy on-call rotation.

[16] "SRE Books," https://sre.google/books/

Chapter 32

Observability

A distributed system is never 100% healthy since, at any given time, there is always something failing. A whole range of failure modes can be tolerated, thanks to relaxed consistency models and resiliency mechanisms like rate limiting, retries, and circuit breakers. But, unfortunately, they also increase the system's complexity. And with more complexity, it becomes increasingly harder to reason about the multitude of emergent behaviors the system might experience, which are impossible to predict up front.

As discussed earlier, human operators are still a fundamental part of operating a service as there are things that can't be automated, like debugging the root cause of a failure. When debugging, the operator makes a hypothesis and tries to validate it. For example, the operator might get suspicious after noticing that the variance of their service's response time has increased slowly but steadily over the past weeks, indicating that some requests take much longer than others. After correlating the increase in variance with an increase in traffic, the operator hypothesizes that the service is getting closer to hitting a constraint, like a resource limit. But metrics and charts alone won't help to validate this hypothesis.

Observability is a set of tools that provide granular insights into a system in production, allowing one to understand its emergent

behaviors. A good observability platform strives to minimize the time it takes to validate hypotheses. This requires granular events with rich contexts since it's impossible to know up front what will be useful in the future.

At the core of observability, we find telemetry sources like *metrics*, *event logs*, and *traces*. Metrics are stored in time-series data stores that have high throughput but struggle with high dimensionality. Conversely, event logs and traces end up in stores that can handle high-dimensional data[1] but struggle with high throughput. Metrics are mainly used for monitoring, while event logs and traces are mainly for debugging.

Observability is a superset of monitoring. While monitoring is focused exclusively on tracking a system's health, observability also provides tools to understand and debug the system. For example, monitoring on its own is good at detecting failure symptoms but less so at explaining their root cause (see Figure 32.1).

32.1 Logs

A *log* is an immutable list of time-stamped events that happened over time. An *event* can have different formats. In its simplest form, it's just free-form text. It can also be structured and represented with a textual format like JSON or a binary one like Protobuf. When structured, an event is typically represented with a bag of key-value pairs:

```
{
  "failureCount": 1,
  "serviceRegion": "EastUs2",
  "timestamp": 1614438079
}
```

Logs can originate from our services or external dependencies, like

[1] Azure Data Explorer is one such event store, see "Azure Data Explorer: a big data analytics cloud platform optimized for interactive, adhoc queries over structured, semi-structured and unstructured data," https://azure.microsoft.com/mediahandler/files/resourcefiles/azure-data-explorer/Azure_Data_Explorer_white_paper.pdf

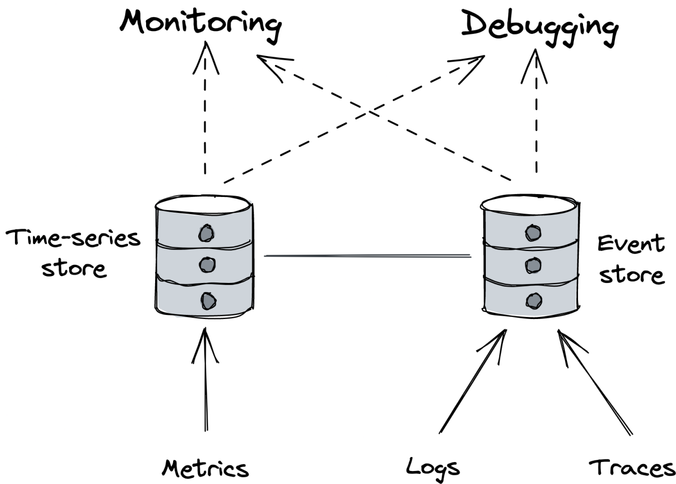

Figure 32.1: Observability is a superset of monitoring.

message brokers, proxies, data stores, etc. Most languages offer libraries that make it easy to emit structured logs. Logs are typically dumped to disk files, which are sent by an agent to an external log collector asynchronously, like an ELK stack[2] or AWS CloudWatch logs.

Logs provide a wealth of information about everything that's happening in a service, assuming it was instrumented properly. They are particularly helpful for debugging purposes, as they allow us to trace back the root cause from a symptom, like a service instance crash. They also help investigate long-tail behaviors that are invisible to metrics summarized with averages and percentiles, which can't explain why a specific user request is failing.

Logs are simple to emit, particularly so free-form textual ones. But that's pretty much the only advantage they have compared to metrics and other telemetry data. Logging libraries can add overhead to our services if misused, especially when they are not asynchronous and block while writing to disk. Also, if the disk fills up due to excessive logging, at best we lose logs, and at worst,

[2] "What is the ELK Stack?," https://www.elastic.co/what-is/elk-stack

the service instance stops working correctly.

Ingesting, processing, and storing massive troves of data is not cheap either, no matter whether we plan to do this in-house or use a third-party service. Although structured binary logs are more efficient than textual ones, they are still expensive due to their high dimensionality.

Finally, but no less importantly, logs have a low signal-to-noise ratio because they are fine-grained and service-specific, making it challenging to extract useful information.

Best practices

To make the job of the engineer drilling into the logs less painful, all the data about a specific *work unit* should be stored in a single event. A work unit typically corresponds to a request or a message pulled from a queue. To effectively implement this pattern, code paths handling work units need to pass around a context object containing the event being built.

An event should contain useful information about the work unit, like who created it, what it was for, and whether it succeeded or failed. It should also include measurements, like how long specific operations took. In addition, every network call performed within the work unit needs to be instrumented and log, e.g., its response time and status code. Finally, data logged to the event should be sanitized and stripped of potentially sensitive properties that developers shouldn't have access to, like users' personal data.

Collating all data within a single event for a work unit minimizes the need for joins but doesn't completely eliminate it. For example, if a service calls another downstream, we will have to perform a join to correlate the caller's event log with the callee's one to understand why the remote call failed. To make that possible, every event should include the identifier of the request (or message) for the work unit.

Costs

There are various ways to keep the costs of logging under con-

trol. A simple approach is to have different logging levels (e.g., debug, info, warning, error) controlled by a dynamic knob that determines which ones are emitted. This allows operators to increase the logging verbosity for investigation purposes and reduce costs when granular logs aren't needed.

Sampling[3] is another tool at our disposal for reducing verbosity. For example, a service could log only every nth event. Additionally, events can also be prioritized based on their expected signal-to-noise ratio: logging failed requests should have a higher sampling frequency than logging successful ones.

The options discussed so far only reduce the logging verbosity on a single node. As we scale out and add more nodes, the logging volume will necessarily increase. Even with the best intentions, someone could check in a bug that leads to excessive logging. To avoid costs soaring through the roof or overloading our log collector service, log collectors need to be able to rate-limit requests.

Of course, we can always decide to create in-memory aggregates (e.g., metrics) from the measurements collected in events and emit just those rather than raw logs. However, by doing so, we trade off the ability to drill down into the aggregates if needed.

32.2 Traces

Tracing captures the entire lifespan of a request as it propagates throughout the services of a distributed system. A *trace* is a list of causally-related spans that represent the execution flow of a request in a system. A *span* represents an interval of time that maps to a logical operation or work unit and contains a bag of key-value pairs (see Figure 32.2).

When a request begins, it's assigned a unique trace ID. The trace ID is propagated from one stage to another at every fork in the local execution flow from one thread to another, and from caller to callee in a network call (through HTTP headers, for example).

[3] "Dynamic Sampling by Example," https://www.honeycomb.io/blog/dynamic-sampling-by-example/

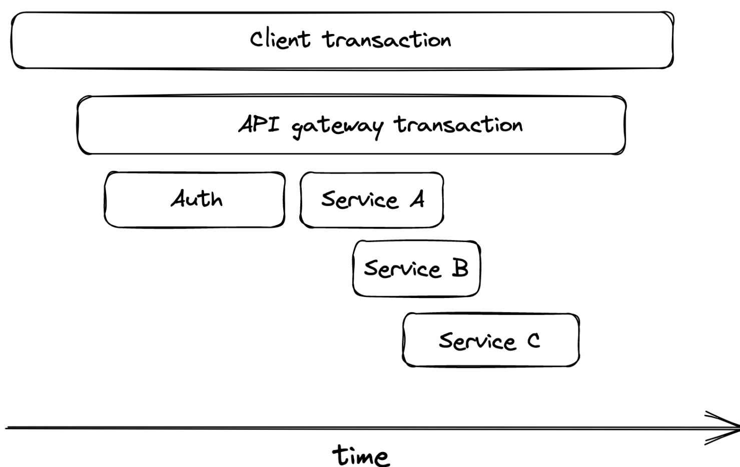

Figure 32.2: An execution flow can be represented with spans.

Each stage is represented with a span — an event containing the trace ID.

When a span ends, it's emitted to a collector service, which assembles it into a trace by stitching it together with the other spans belonging to the same trace. Popular distributed tracing collectors include Open Zipkin[4] and AWS X-ray[5].

Traces allow developers to:

- debug issues affecting very specific requests, which can be used to investigate failed requests raised by customers in support tickets;
- debug rare issues that affect only an extremely small fraction of requests;
- debug issues that affect a large fraction of requests that have something in common, like high response times for requests that hit a specific subset of service instances;
- identify bottlenecks in the end-to-end request path;
- identify which users hit which downstream services and

[4] "Zipkin: a distributed tracing system," https://zipkin.io/
[5] "AWS X-Ray," https://aws.amazon.com/xray/

in what proportion (also referred to as *resource attribution*), which can be used for rate-limiting or billing purposes.

Tracing is challenging to retrofit into an existing system since it requires every component in the request path to be modified to propagate the trace context from one stage to the other. And it's not just the components that are under our control that need to support tracing; third-party frameworks, libraries, and services need to as well.[6].

32.3 Putting it all together

The main drawback of event logs is that they are fine-grained and service-specific. When a user request flows through a system, it can pass through several services. A specific event only contains information for the work unit of one specific service, so it can't be of much use for debugging the entire request flow. Similarly, a single event doesn't give much information about the health or state of a specific service.

This is where metrics and traces come in. We can think of them as abstractions, or derived views, built from event logs and optimized for specific use cases. A metric is a time series of summary statistics derived by aggregating counters or observations over multiple events. For example, we could emit counters in events and have the backend roll them up into metrics as they are ingested. In fact, this is how some metric-collection systems work.

Similarly, a trace can be derived by aggregating all events belonging to the lifecycle of a specific user request into an ordered list. Just like in the previous case, we can emit individual span events and have the backend aggregate them together into traces.

[6]The service mesh pattern can help retrofit tracing.

Chapter 33

Manageability

Operators use observability tools to understand the behavior of their application, but they also need a way to modify the behavior without code changes. One example of this is releasing a new version of an application to production, which we discussed earlier. Another example is changing the way an application behaves by modifying its configuration.

An application generally depends on a variety of configuration settings. Some affect its behavior, like the maximum size of an internal cache, while others contain secrets, like credentials to access external data stores. Because settings vary by environment and can contain sensitive information, they should not be hardcoded.

To decouple the application from its configuration, the configuration can be persisted in a dedicated store[1] like AWS AppConfig[2] or Azure App Configuration[3].

At deployment time, the CD pipeline can read the configuration from the store and pass it to the application through environment

[1] "Continuous Configuration at the Speed of Sound," https://www.allthingsdistributed.com/2021/08/continuous-configuration-on-aws.html

[2] "AWS AppConfig Documentation," https://docs.aws.amazon.com/appconfig/

[3] "Azure App Configuration," https://azure.microsoft.com/en-us/services/app-configuration/

variables. The drawback of this approach is that the configuration cannot be changed without redeploying the application.

For the application to be able to react to configuration changes, it needs to periodically re-read the configuration during run time and apply the changes. For example, if the constructor of an HTTP handler depends on a specific configuration setting, the handler needs to be re-created when that setting changes.

Once it's possible to change configuration settings dynamically, new features can be released with settings to toggle (enable/disable) them. This allows a build to be released with a new feature disabled at first. Later, the feature can be enabled for a fraction of application instances (or users) to build up confidence that it's working as intended before it's fully rolled out. Similarly, the same mechanism can be used to perform A/B tests[4].

[4]"A/B testing," https://en.wikipedia.org/wiki/A/B_testing

Summary

If you have spent even a few months in a team that operates a production service, you should be very familiar with the topics discussed in this part. Although we would all love to design new large-scale systems and move on, the reality is that the majority of the time is spent in maintenance activities, such as fixing bugs, adding new features to existing services, and operating production services.

In my opinion, the only way to become a better system designer is to embrace these maintenance activities and aspire to make your systems easy to modify, extend and operate so that they are easy to maintain. By doing so, you will pick up a "sixth sense" that will allow you to critique the design of third-party systems and ultimately make you a better systems designer/builder.

So my advice to you is to use every chance you get to learn from production services. Be on call for your services. Participate in as many post-mortems as you can and relentlessly ask yourself how an incident could have been avoided. These activities will pay off in the long term a lot more than reading about the latest architectural patterns and trends.

Chapter 34

Final words

Congratulations, you reached the end of the book! I hope you learned something you didn't know before and perhaps even had a few "aha" moments. Although this is the end of the book, it's just the beginning of your journey.

The best way to continue your learning journey is to go out in the world and design, build and maintain large-scale systems. Some of the best-paid jobs in the industry involve building such systems, and there is no shortage of demand. The second-best way is by reading industry papers, which provide a wealth of knowledge about distributed systems that have stood the test of time. And after reading this book, these papers should be accessible to you.

For example, you could start with "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency"[1], which describes Azure's cloud storage system. Azure's cloud storage is the core building block on top of which Microsoft built many other successful products. One of the key design decisions was to guarantee strong consistency, making the application developers' job much easier. I like this paper a lot because it touches upon many of the concepts discussed in this book.

[1] "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency," https://sigops.org/s/conferences/sosp/2011/current/2011-Cascais/printable/11-calder.pdf

Once you have digested the Azure storage paper, I suggest reading "Azure Data Explorer: a big data analytics cloud platform optimized for interactive, ad-hoc queries over structured, semi-structured and unstructured data"[2]. This paper discusses the implementation of a cloud-native event store built on top of Azure's cloud storage — a great example of how these large-scale systems compose on top of each other.

If any concept is new to you or unclear, follow your curiosity down the rabbit hole by exploring referenced papers.[3]. As a bonus exercise, this book is sprinkled with great papers and references. Read them all to complete all the side quests!

In terms of book recommendations, Martin Kleppman's "Designing Data-Intensive Applications"[4] is a must-read. The book describes the pros and cons of various technologies for processing and storing data. Also, for some more high-level case studies, check out Alex Xu's "System Design Interview"[5] — it's a good read even if you aren't preparing for interviews.

As no book is ever perfect, I'm always happy to receive feedback. So if you have found an error, have an idea for improvement, or simply want to comment on something, always feel free to write me[6]. I love connecting with readers.

Finally, please consider leaving a review on Goodreads or Amazon if you liked the book. It makes all the difference to prospective readers.

[2] "Azure Data Explorer: a big data analytics cloud platform optimized for interactive, ad-hoc queries over structured, semi-structured and unstructured data," https://azure.microsoft.com/mediahandler/files/resourcefiles/azure-data-explorer/Azure_Data_Explorer_white_paper.pdf

[3] If you are new to reading papers, use the three-pass approach by S. Keshav, see "How to Read a Paper," https://web.stanford.edu/class/ee384m/Handouts/HowtoReadPaper.pdf

[4] "Designing Data-Intensive Applications," https://www.amazon.com/dp/1449373321

[5] "System Design Interview," https://www.amazon.co.uk/dp/B08CMF2CQF

[6] roberto@understandingdistributed.systems