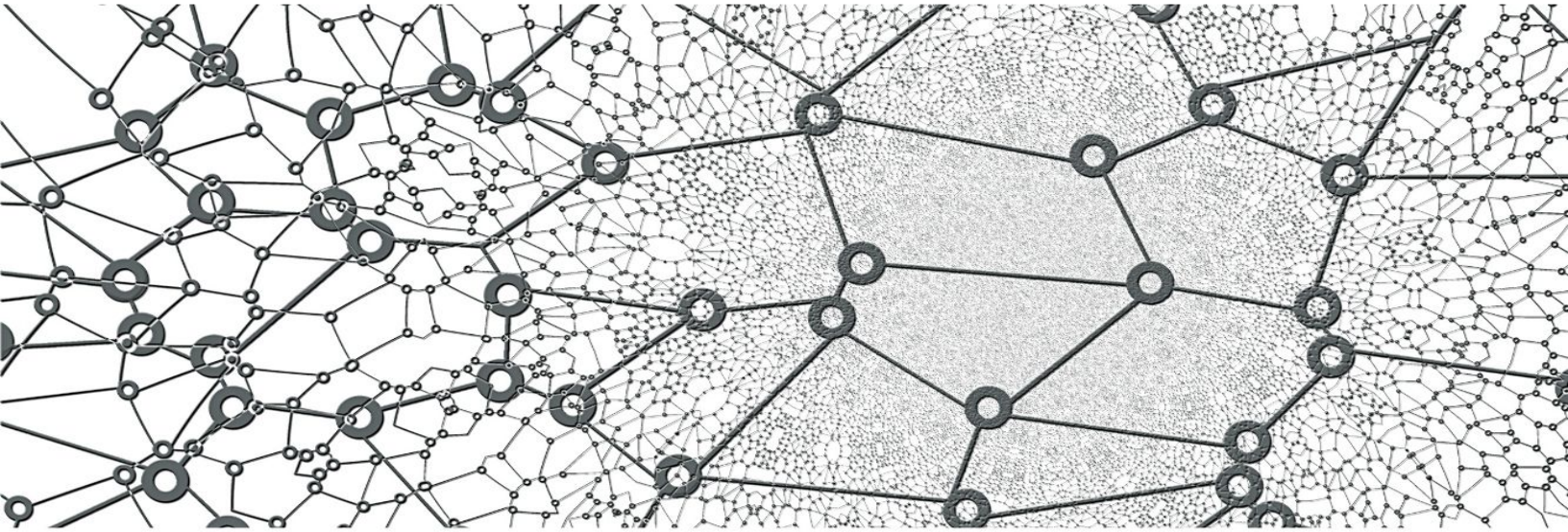
Roberto Vitillo

UNDERSTANDING DISTRIBUTED SYSTEMS



WHAT EVERY DEVELOPER SHOULD KNOW ABOUT LARGE DISTRIBUTED APPLICATIONS

Understanding Distributed Systems

Version 1.0.4

Roberto Vitillo

February 2021

Understanding Distributed Systems

Copyright

The book's diagrams have been created with Excalidraw.

About the author

Authors generally write this page in the third person as if someone else is writing about them. 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.

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

Before that, 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.

After getting my master's degree in computer science, 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.

Acknowledgements

Writing a book is an incredibly challenging but rewarding experience. I wanted to share what I have learned about distributed systems for a very long time.

I appreciate the colleagues who inspired and believed in me. Thanks to Chiara Roda, Andrea Dotti, Paolo Calafiura, Vladan Djeric, Mark Reid, Pawel Chodarcewicz, and Nuno Cerqueira.

Doug Warren, Vamis Xhagjika, Gaurav Narula, Alessio Placitelli, Kofi Sarfo, Stefania Vitillo and Alberto Sottile were all kind enough to provide invaluable feedback. Without them, the book wouldn't be what it is today.

Finally, and above all, thanks to my family: Rachell and Leonardo. You always believed in me. That made all the difference.

Preface

According to Stack Overflow's 2020 developer survey, the best-paid engineering roles require distributed systems expertise. That comes as no surprise as modern applications are distributed systems.

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.

When I first started learning about distributed systems, I spent hours connecting the missing dots between theory and practice. I was looking for an accessible and pragmatic introduction to guide me through the maze of information and setting me on the path to becoming a practitioner. But there was nothing like that available.

That is why I decided to write a book to teach the fundamentals of distributed systems so that you don't have to spend countless hours scratching your head to understand how everything fits together. This is the guide I wished existed when I first started out, and it's based on my experience building large distributed systems that scale to millions of requests per second and billions of devices.

I plan to update the book regularly, which is why it has a version number. You can subscribe to receive updates from the book's landing page. 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[1].

0.1 Who should read this book

If you develop the back-end of web or mobile applications (or would like to!), this book is for you. When building distributed systems, you need to be familiar with the network stack, data consistency models, scalability and reliability patterns, and much more. Although you can build applications without knowing any of that, you will end up spending hours debugging and re-designing their architecture, learning lessons that you could have acquired in a much faster and less painful way. Even if you are an experienced engineer, this book will help you fill gaps in your knowledge that will make you a better practitioner and system architect.

The book also makes for a great study companion for a system design interview if you want to land a job at a company that runs large-scale distributed systems, like Amazon, Google, Facebook, or Microsoft. If you are interviewing for a senior role, you are expected to be able to design complex networked services and dive deep into any vertical. You can be a world champion at balancing trees, but if you fail the design round, you are out. And if you just meet the bar, don't be surprised when your offer is well below what you expected, even if you aced everything else.

---

1. roberto@understandingdistributed.systems↩

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 composed of nodes that cooperate to achieve some task by exchanging messages over communication links. A node can generically refer to a physical machine (e.g., a phone) or a software process (e.g., 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. Dropbox replicates your data across multiple nodes so that the loss of a single node doesn't cause all 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 hundreds 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 with high resolutions because it has a datacenter close to you.

This book will guide you through the fundamental challenges that need to be solved to design, build and operate distributed systems: communication,

coordination, scalability, resiliency, and operations.

1.1 Communication

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

How are 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 can you guarantee that no intermediary can snoop into 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, and you need to understand how the stack works when that happens.

1.2 Coordination

Another hard challenge of building distributed systems is coordinating nodes into a single coherent whole in the presence of failures. A fault is a component that stopped working, and a system is fault-tolerant when it can continue to operate despite one or more faults. The "two generals" problem is a famous thought experiment that showcases why this is a challenging problem.

Suppose there are two generals (nodes), each commanding its own army, that need to agree on a time to jointly attack a city. There is some distance between the armies, and the only way to communicate is by sending a messenger (messages). Unfortunately, these messengers can be captured by the enemy (network failure).

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 and wait for a response.

What if no response arrives, though? Was one of the messengers captured? Perhaps a messenger was injured, and it's taking longer than expected to arrive at the destination? Should the general send another messenger?

You can see that this problem is much harder than it originally appeared. As it turns out, no matter how many messengers are dispatched, neither general can be completely certain that the other army will attack the city at the same time. Although sending more messengers increases the general's confidence, it never reaches absolute certainty.

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

1.3 Scalability

The performance of a distributed system represents how efficiently it handles load, and it's generally measured with *throughput* and *response time*. Throughput is the number of operations processed per second, and response time is the total time between a client request and its response.

Load can be measured in different ways since it's specific to the system's use cases. For example, number of concurrent users, number of communication links, or ratio of writes to reads are all different forms of load.

As the load increases, it will eventually reach the system's *capacity* — the maximum load the system can withstand. At that point, the system's performance either plateaus or worsens, 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 timeout.

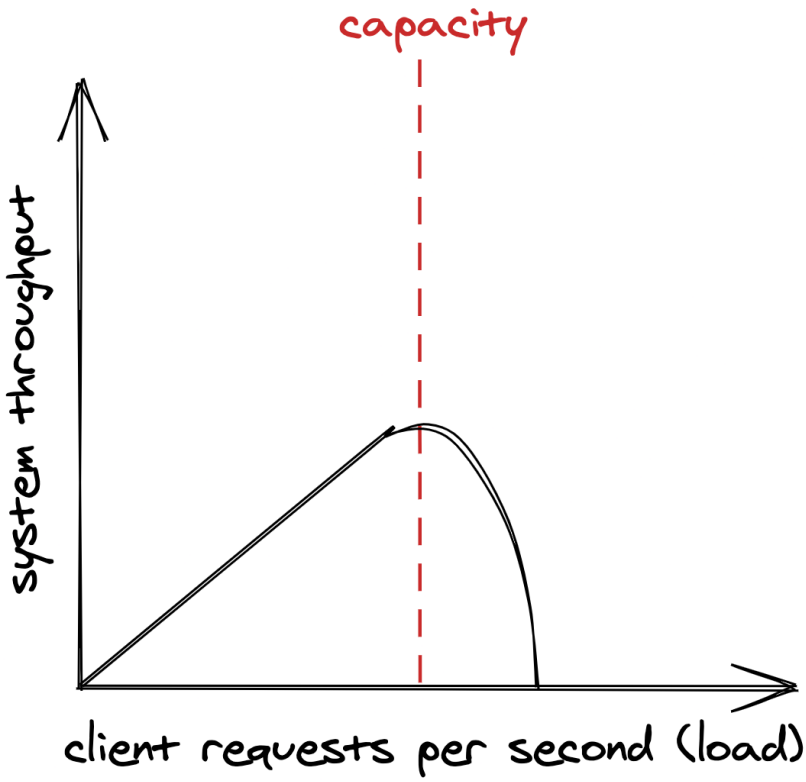


Figure 1.1: The system throughput on the y axis is the subset of client requests (x axis) that can be handled without errors and with low response times, also referred to as its goodput.

The capacity of a distributed system depends on its architecture and an intricate web of physical limitations like the nodes' memory size and clock cycle, and the bandwidth and latency of network links.

A quick and easy way to increase the capacity is buying more expensive hardware with better performance, which is referred to as *scaling up*. But that will hit a brick wall sooner or later. When that option is no longer available, the alternative is *scaling out* by adding more machines to the system.

In the book's third part, we will explore the main architectural patterns that you can leverage to scale out applications: functional decomposition, duplication, and partitioning.

1.4 Resiliency

A distributed system is resilient when it can continue to do its job even when failures happen. And at scale, any failure that can happen will eventually occur. Every component of a system 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 absolute number of failures becomes. And it gets worse, since failures typically are not independent, the failure of a component can increase the probability that another one will fail.

Failures that are left unchecked can impact the system's *availability*, which is defined as the amount of time the application can serve requests divided by the duration of the period measured. In other words, it's the percentage of time the system is capable of servicing requests and doing useful work.

Availability is often described with nines, a shorthand way of expressing percentages of availability. Three nines are typically considered acceptable, 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, which only increase as the application scales out to handle more load, its availability will inevitably drop. Because of that, a distributed system needs to embrace failure and work around it using techniques such as redundancy and self-healing mechanisms.

As an engineer, you need to be paranoid and assess the risk that a component can fail by considering the likelihood of it happening and its resulting impact when it does. If the risk is high, you will need to mitigate

it. Part 4 of the book is dedicated to fault tolerance and it introduces various resiliency patterns, such as rate limiting and circuit breakers.

1.5 Operations

Distributed systems need to be tested, deployed, and maintained. It used to be that one team developed an application, and another was responsible for operating it. The rise of microservices and DevOps has changed that. The same team that designs a system is also responsible for its live-site operation. That's a good thing as there is no better way to find out where a system falls short than experiencing it by being on-call for it.

New deployments need to be rolled out continuously in a safe manner without affecting the system's availability. The system needs to be observable so that it's easy to understand what's happening at any time. Alerts need to fire when its service level objectives are at risk of being breached, and a human needs to be looped in. The book's final part explores best practices to test and operate distributed systems.

1.6 Anatomy of a distributed system

Distributed systems come in all shapes and sizes. The book anchors the discussion to the backend of systems composed of commodity machines that work in unison to implement a business feature. This comprises the majority of large scale systems being built today.

Before we can start tackling the fundamentals, we need to discuss the different ways a distributed system can be decomposed into parts and relationships, or in other words, its architecture. The architecture differs depending on the angle you look at it.

Physically, a distributed system is an ensemble of physical machines that communicate over network links.

At run-time, a distributed system is composed of software processes that communicate via *inter-process communication* (IPC) mechanisms like HTTP, and are hosted on machines.

From an implementation perspective, a distributed system is a set of loosely-coupled components that can be deployed and scaled independently called services.

A *service* implements one specific part of the overall system's capabilities. At the core of its implementation is the business logic, which exposes interfaces used to communicate with the outside world. By interface, I mean the kind offered by your language of choice, like Java or C#. An "inbound" interface defines the operations that a service offers to its clients. In contrast, an "outbound" interface defines operations that the service uses to communicate with external services, like data stores, messaging services, and so on.

Remote clients can't just invoke an interface, which is why *adapters* are required to hook up IPC mechanisms with the service's 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 inbound interfaces. In contrast, outbound adapters implement the service's outbound interfaces, granting the business logic access to external services, like data stores. This is illustrated in Figure 1.2.

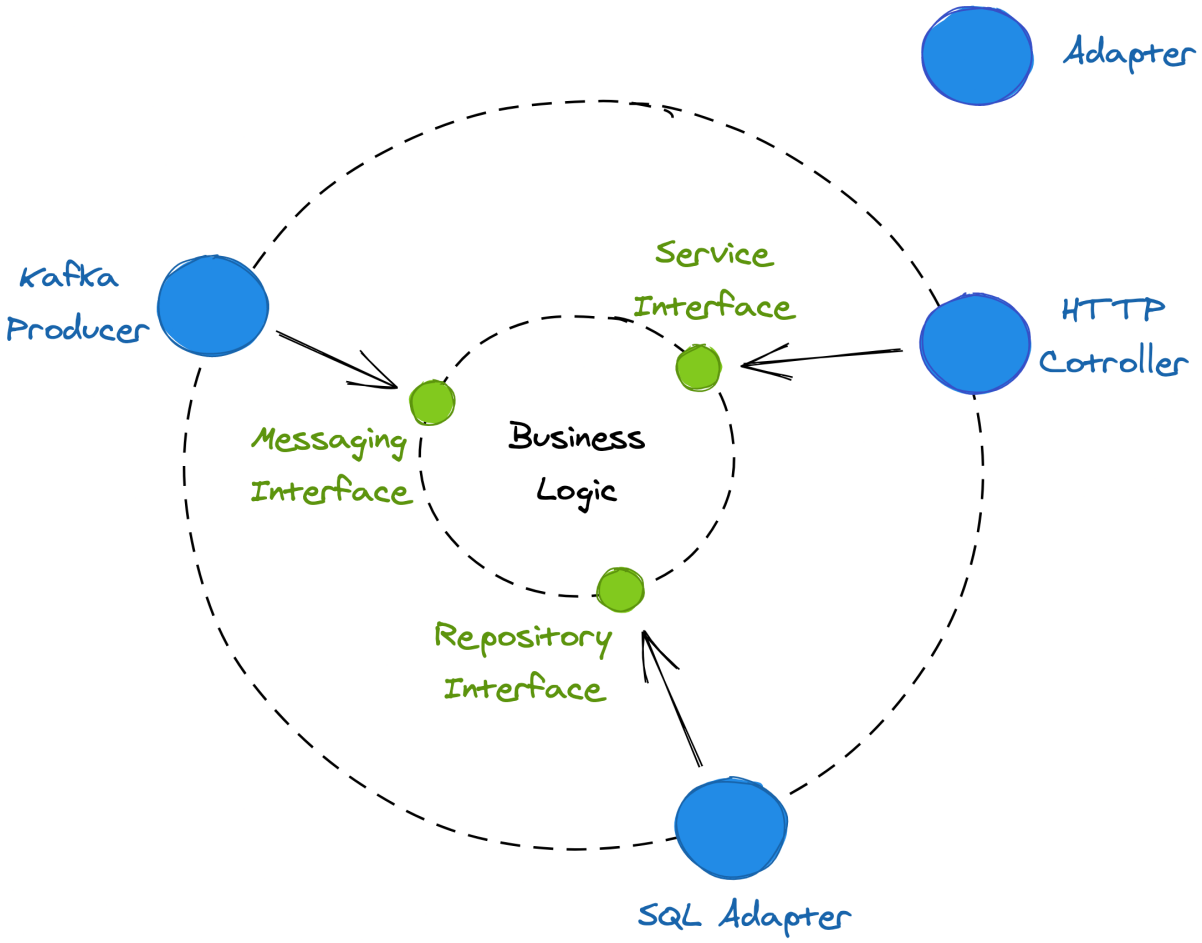


Figure 1.2: The business logic uses the messaging interface implemented by the Kafka producer to send messages and the repository interface to access the SQL store. In contrast, the HTTP controller handles incoming requests using the service interface.

A process running a service is referred to as a *server*, while a process that sends requests to a server is referred to as a *client*. Sometimes, a process is both a client and a server, since the two aren't mutually exclusive.

For simplicity, I will assume that an individual instance of a service runs entirely within the boundaries of a single server process. Similarly, I assume that a process has a single thread. This allows me to neglect some implementation details that only complicate our discussion without adding much value.

In the rest of the book, I will switch between the different architectural points of view (see Figure 1.3), depending on which one is more appropriate to discuss a particular topic. Remember that they are just different ways to look at the same system.

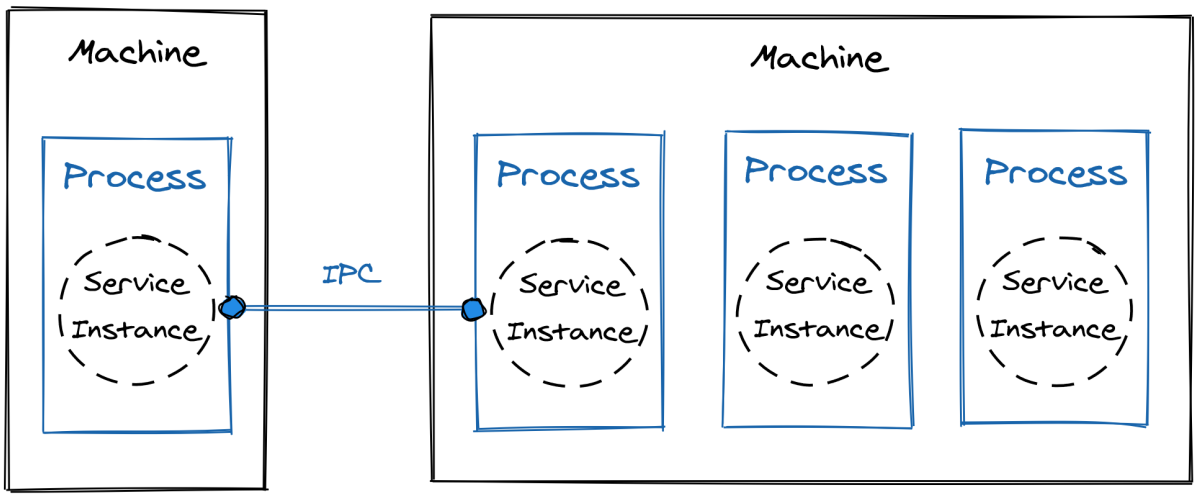


Figure 1.3: The different architectural points of view used in this book.

(PART) Communication

Introduction

Communication between processes over the network, or *inter-process communication* (IPC), is at the heart of distributed systems. Network protocols are arranged in a stack, 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, it moves through the stack from the top layer to the bottom one and vice-versa on the other end, as shown in Figure 1.4.

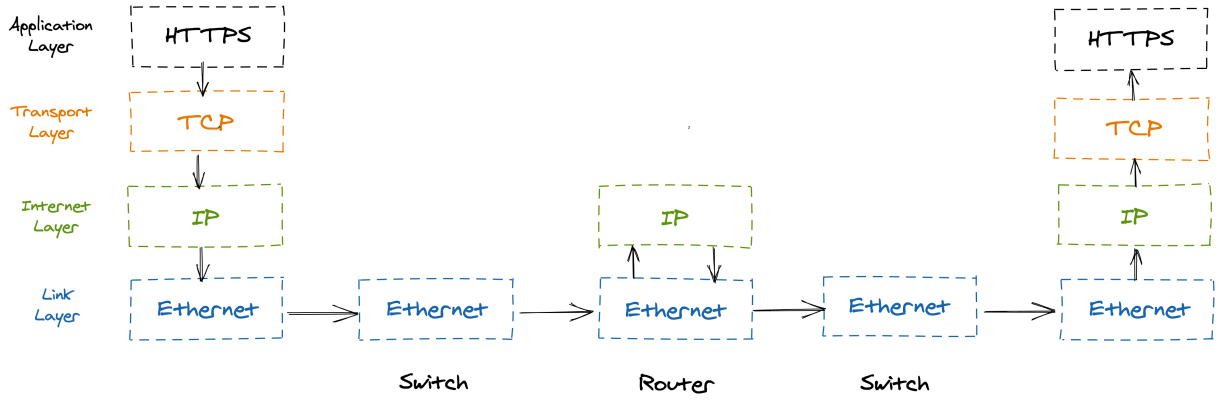


Figure 1.4: Internet protocol suite

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.

The *internet layer* uses addresses to route 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 basis. Routers operate at this layer and forward IP packets based on their destination IP address.

The *transport layer* transmits data between two processes using port numbers to address the processes on either end. The most important protocol in this layer is the Transmission Control Protocol (TCP).

The *application layer* defines high-level communication protocols, like HTTP or DNS. Typically your code will target this level of abstraction.

Even though each protocol builds up on top of the other, sometimes the abstractions leak. If you don't know how the bottom layers work, you will have a hard time troubleshooting networking issues that will inevitably arise.

Chapter 2 describes how to build a reliable communication channel (TCP) on top of an unreliable one (IP), which can drop, duplicate and deliver data out of order. Building reliable abstractions on top of unreliable ones is a common pattern that we will encounter many times as we explore further how distributed systems work.

Chapter 3 describes how to build a secure channel (TLS) on top of a reliable one (TCP), which provides encryption, authentication, and integrity.

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 a first taste of eventually consistency.

Chapter 5 concludes this part by discussing how services can expose APIs that other nodes can use to send commands or notifications to. Specifically, we will dive into the implementation of a RESTful HTTP API.

2 Reliable links

TCP is 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 any gaps, duplication or corruption. TCP also implements a set of stability patterns to avoid overwhelming the network or 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 state of the connection is managed by the operating system 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.

This is a simplification, though, as there are more states 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.
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.

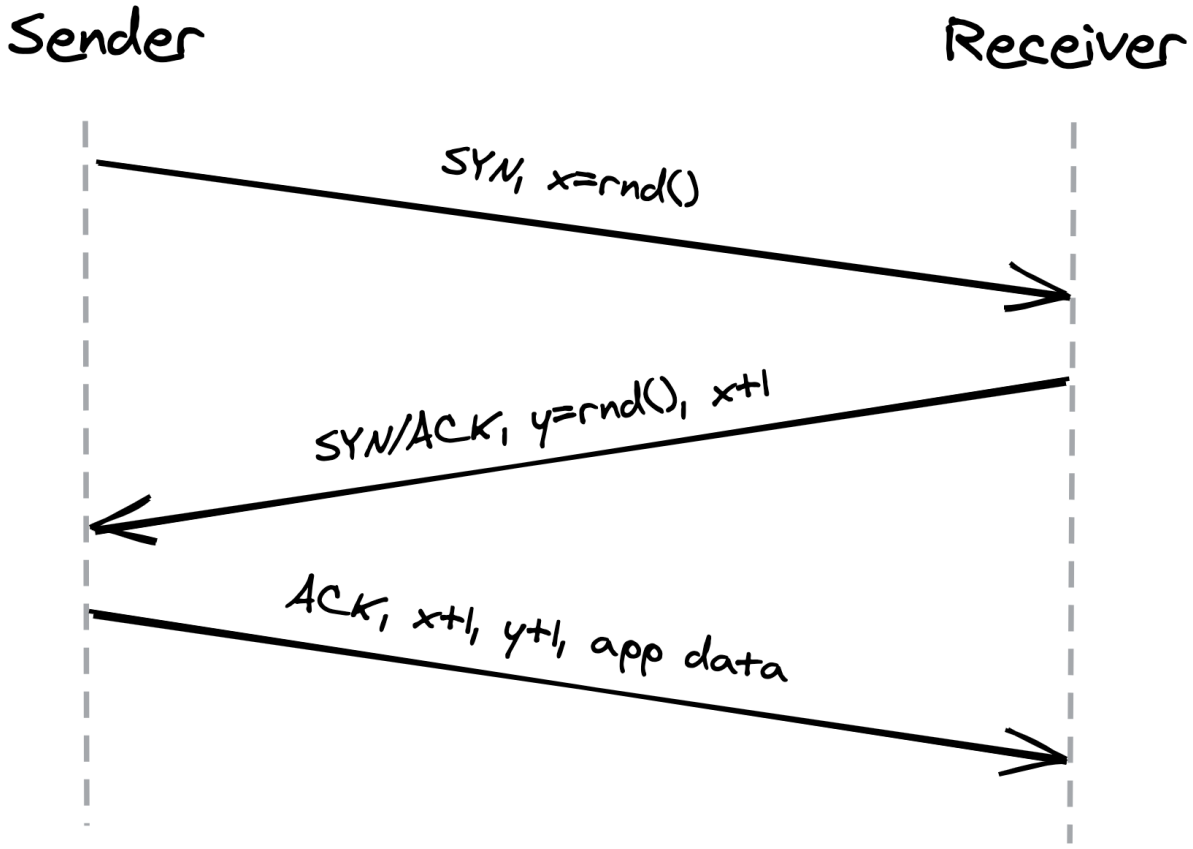


Figure 2.1: Three-way handshake

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

After data transmission is complete, the connection needs to be closed to release all resources on both ends. This termination phase involves multiple round-trips.

2.3 Flow control

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

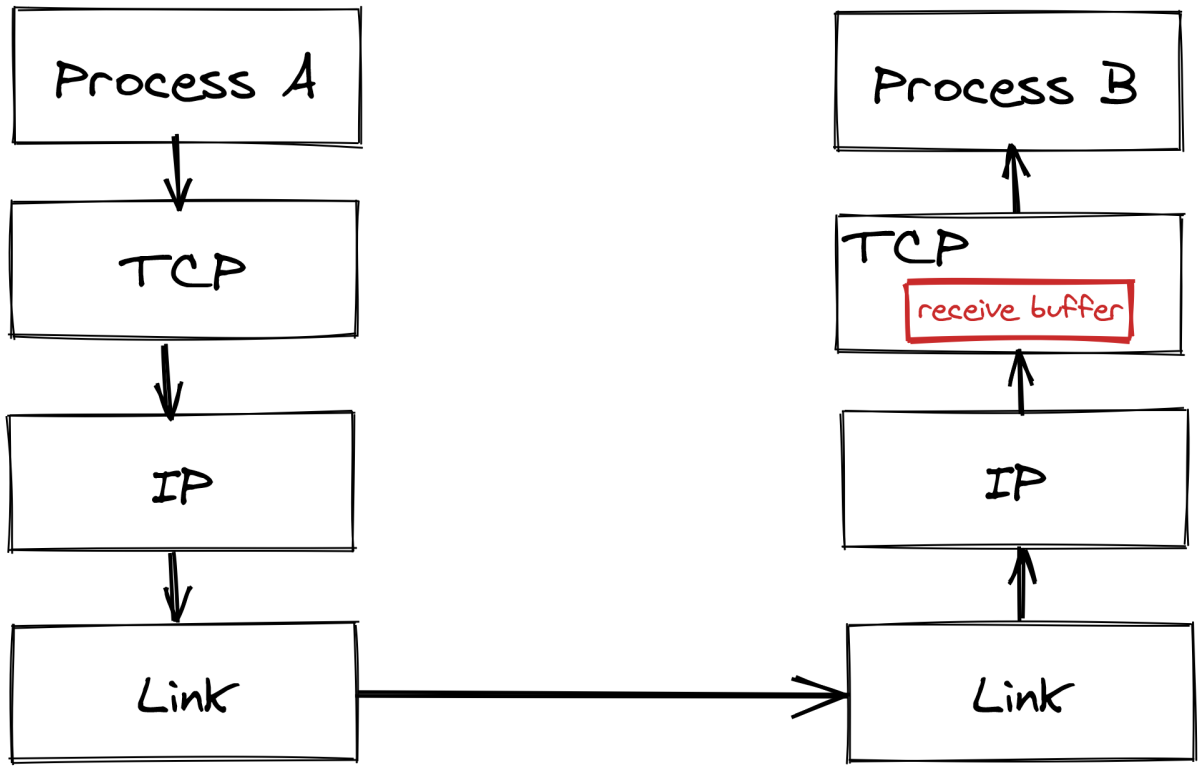


Figure 2.2: The receive buffer stores data that hasn't been processed yet by the application.

The receiver also communicates back to the sender the size of the buffer whenever it acknowledges a segment, as shown in Figure 2.3. The sender, if it's respecting the protocol, avoids sending more data that can fit in the receiver's buffer.

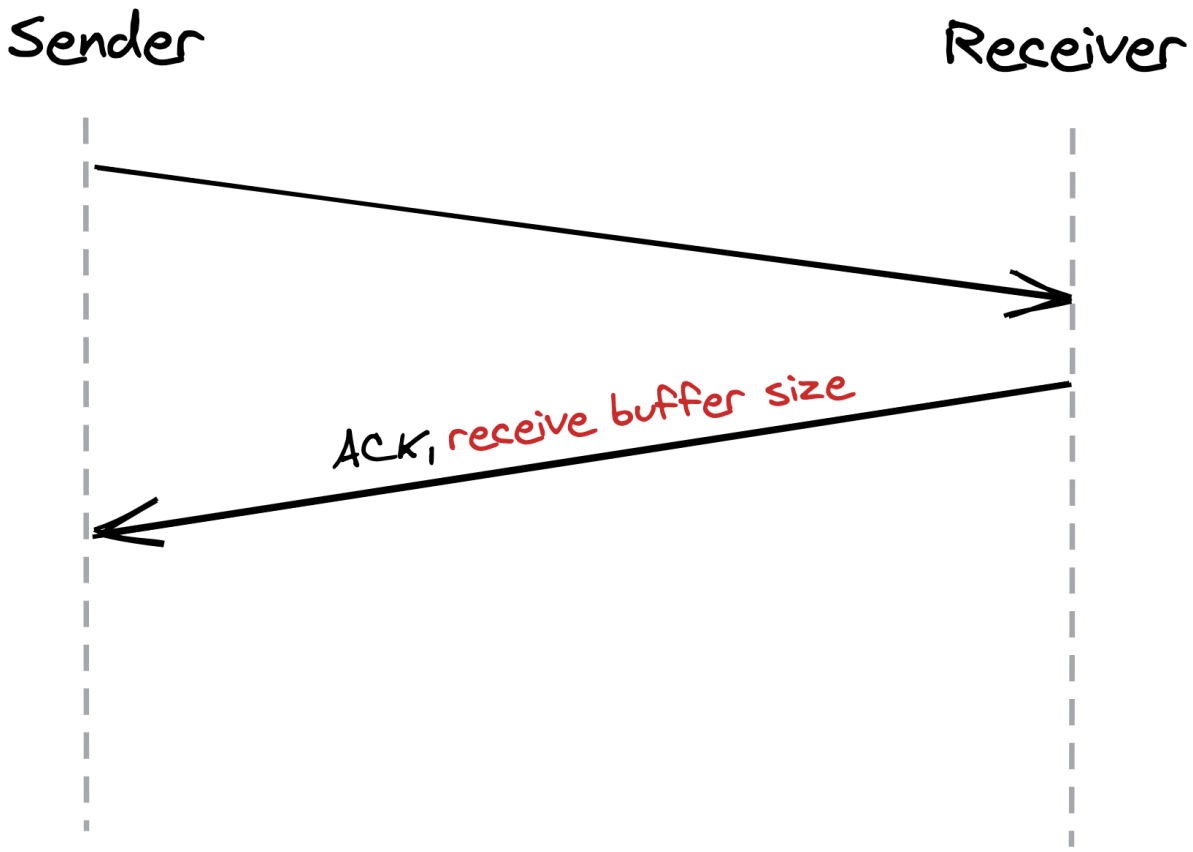


Figure 2.3: The size of the receive buffer is communicated in the headers of acknowledgments segments.

This mechanism is not too dissimilar to rate-limiting at the service level. 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 not only guards against overwhelming the receiver, but also against flooding the underlying network.

The sender estimates the available bandwidth of the underlying network empirically through measurements. 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 size of the receiver window limits the maximum size of the congestion window. 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 reaching an upper limit. This means that we can't use the network's full capacity right after a connection is established. The lower the round trip time (RTT) is, the quicker the sender can start utilizing the underlying network's bandwidth, as shown in Figure 2.4.

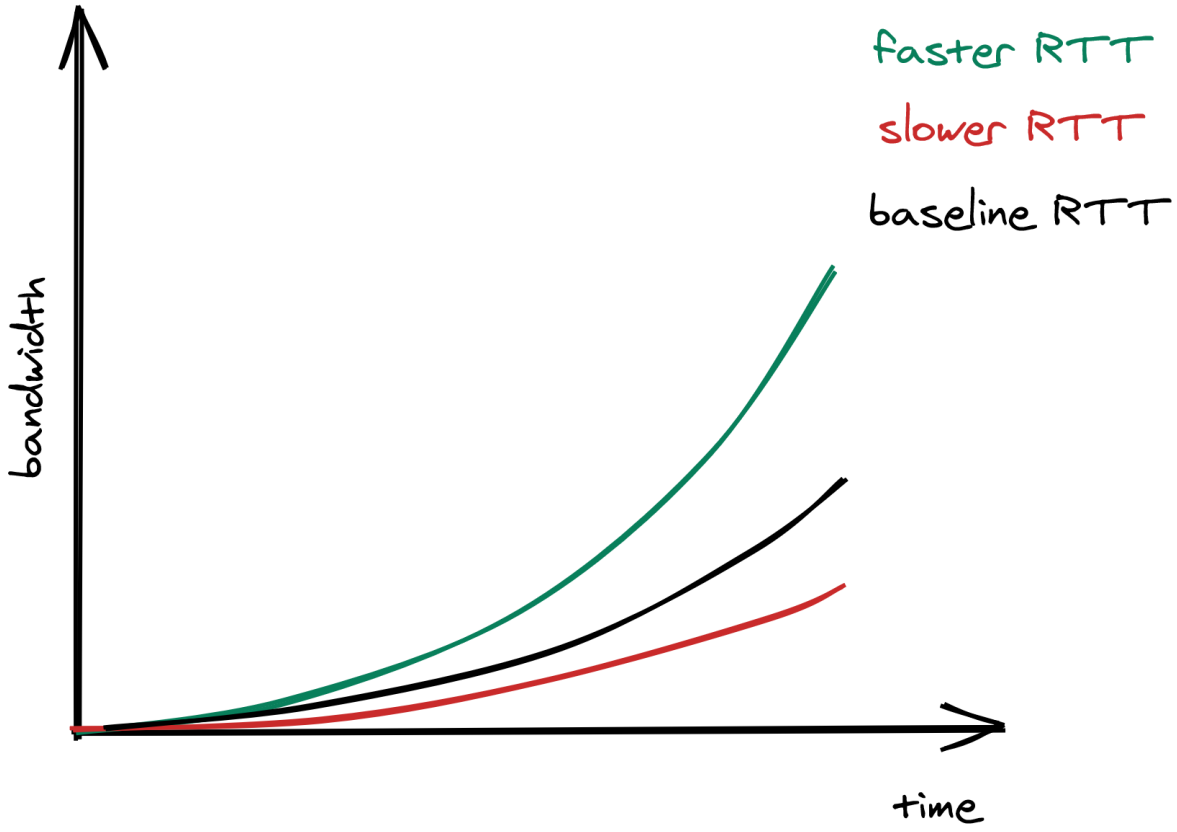


Figure 2.4: The lower the RTT is, the quicker the sender can start utilizing the underlying network's bandwidth.

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 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 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 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 lower the round trip time is, 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 is actually capable of delivering. If you drop the stability and reliability mechanisms that TCP provides, what you get is a simple protocol named *User Datagram Protocol* (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. Clients can only send discrete packets, called datagrams, with a limited size. 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 barebone protocol. It's used to bootstrap custom protocols, which provide some, but not all, of the stability and reliability guarantees that TCP does[1].

For example, in modern multi-player games, clients sample gamepad, mouse and keyboard 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, as TCP would attempt to redeliver the missing data and consequently slow down the client's experience.

---

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

3 Secure links

We now know how to reliably send bytes from one process to another over the network. The problem is these bytes are sent in the clear, and any middle-man can intercept our communication. To protect against that, we can use the *Transport Layer Security* (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*. Both parties generate a key-pair consisting of a private and public part. The processes are then able to create a shared secret by exchanging their public keys. This is possible thanks to some mathematical properties 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 actually come with cryptographic instructions. Unless you have a very good reason, you should use TLS for all communications, even those that are 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 public key of the server to verify that the digital signature was actually signed with the private key. This is possible thanks to mathematical properties of the key-pair.

The problem with this naive approach is that the client has no idea whether the public key shared by the server is authentic, so we have certificates to prove the ownership of a public key for a specific entity. 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 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 CA's, such as Let's Encrypt, are typically included in the client's trusted store by default by the operating system vendor.

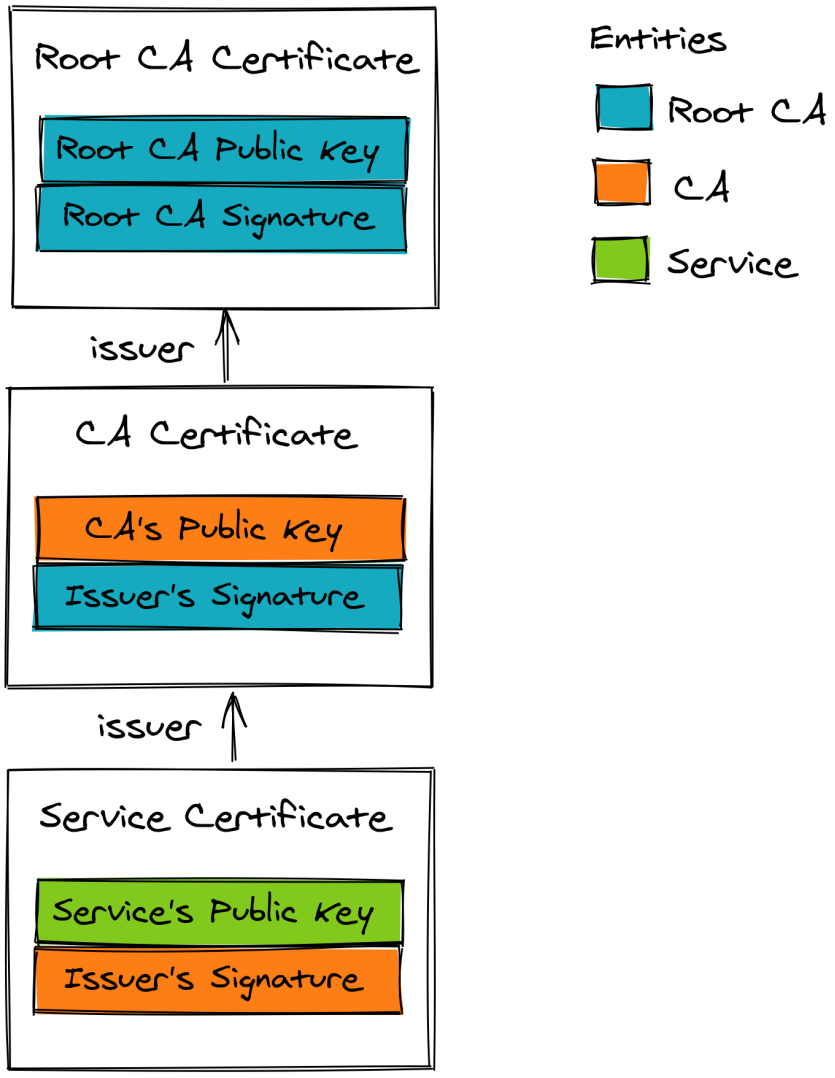


Figure 3.1: A certificate chain ends with a self-signed certificate issued by a root CA.

When a TLS connection is opened, the server sends the full certificate chain to the client, starting with the server's certificate and ending with the root CA. The client verifies the server's certificate by scanning the certificate chain until a certificate is found that it trusts. Then the certificates are verified in the 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 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 service down as clients are no longer able to connect with it. 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 middle man 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 (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 because it fails to detect errors for roughly 1 in 16 million to 10 billion packets. With packets of 1KB, this can happen every 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 negotiated key exchange algorithm to create a shared secret. The shared secret is used by the chosen symmetric encryption algorithm to encrypt the communication of the secure channel going forwards.
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. The handshake typically requires 2 round trips with TLS 1.2 and just one with TLS 1.3. The bottom line is creating a new connection is expensive; yet another reason to put your servers geographically closer to the clients and reuse connections when possible.

4 Discovery

So far, we explored how to create a reliable and secure channel between two processes located on different machines. However, to create a new connection with a remote process, we still need to discover its IP address. To resolve hostnames into IP addresses, we can use the phone book of the Internet: the *Domain Name System* (DNS) — a distributed, hierarchical, and eventually consistent key-value store.

In this chapter, we will look at how DNS resolution works in a browser, but the process is the same for any other client. 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.

Concretely, let's take a look at how the DNS resolution works when you type *www.example.com* in your browser (see Figure 4.1).

1. The browser checks whether it has resolved the hostname before in its local cache. If so, it returns the cached IP address; otherwise it routes the request to a DNS resolver. The DNS resolver is typically a DNS server hosted by your Internet Service Provider.

2. The resolver is responsible for iteratively translating the hostname for the client. The reason why it's iterative will become evident 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 an incoming request, like *.com*, to the name server's address responsible for it.

4. The resolver, armed with the address of the TLD, sends the resolution request to the TLD name server for the domain, in our case *.com*.

5. The TLD name server maps the domain name of a request to the address of the *authoritative name server* responsible for it. An

authoritative name server is responsible for a specific domain and holds all records that map the hostnames to IP addresses within that domain.

6. The resolver finally queries the authoritative name server for www.example.com, which checks its entries for the *www* hostname and returns the IP address associated with it back to the resolver.

If the query included a subdomain of *example.com*, like, e.g., *news.example.com*, the authoritative name server would have returned the address of the name server responsible for the subdomain.

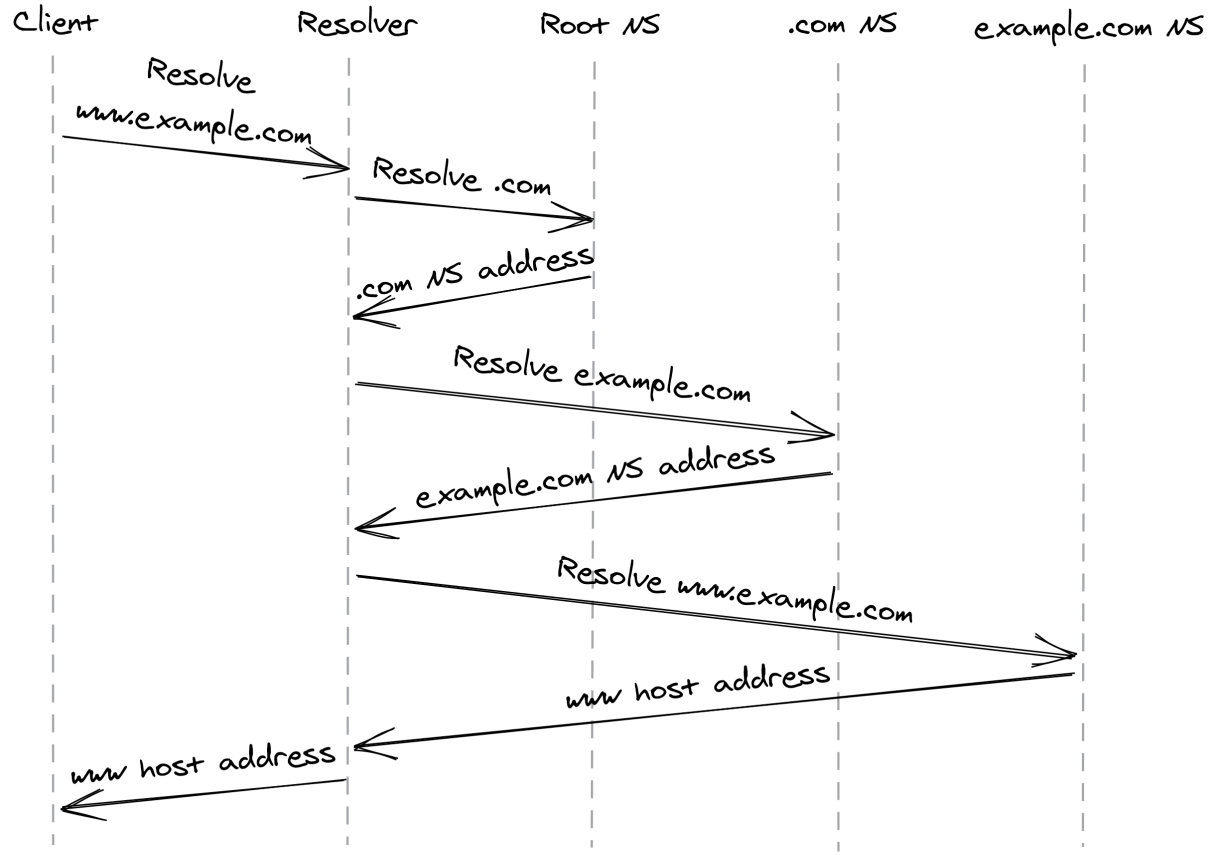


Figure 4.1: DNS resolution process

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 any hostname. Given the costs involved resolving a hostname, it

comes as no surprise that the designers of DNS thought of ways to reduce them.

DNS uses UDP to serve DNS queries as it's lean and has a low overhead. UDP at the time was a great choice as there is no price to be paid to open a new connection. That said, it's not secure, as requests are sent in the clear over the Internet, allowing third parties to snoop in. Hence, the industry is pushing slowly towards running DNS on top of TLS.

The resolution would be slow if every request had to go through several name server lookups. Not only that, but think of the scale requirements on the name servers to handle the global resolution load. Caching is used to speed up the resolution process, as 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. But, there is no guarantee that the client plays nicely and enforces the TTL. Don't be surprised when you change a DNS entry and find out that a small fraction of clients are still trying to connect to the old address days after the change.

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 the clients will have to resolve the entry more often.

If your name server becomes unavailable for any reason, the smaller the record's TTL is and 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 the clients can't find the IP address of your service, they won't have a way to connect it. This can lead to massive outages.

5 APIs

A service exposes operations to its consumers via a set of interfaces implemented by its business logic. As remote clients can't access these directly, adapters — which make up the service's application programming interface (API) — translate messages received from IPC mechanisms to interface calls, as shown in Figure 5.1.

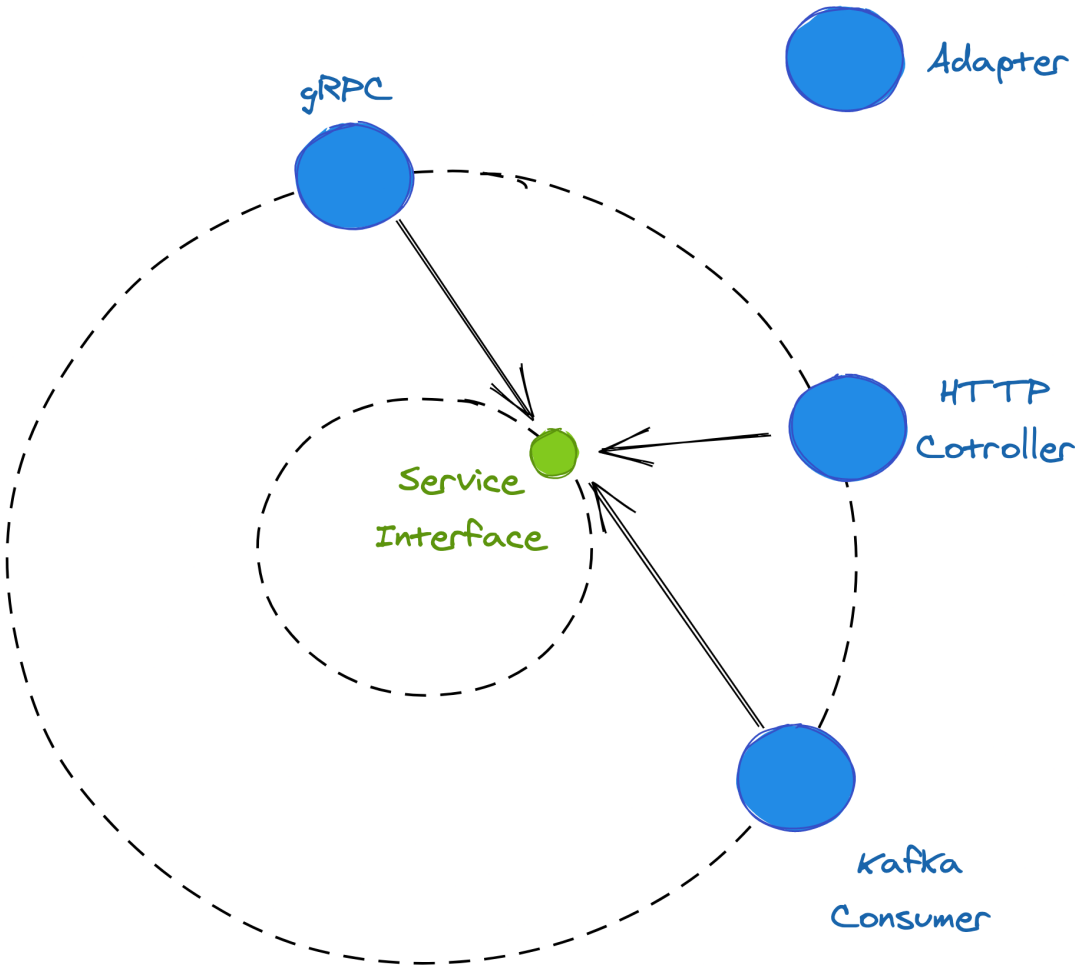


Figure 5.1: Adapters translate messages received from IPC mechanisms to interface calls.

The communication style between a client and a service can be *direct* or *indirect*, depending on whether the client communicates directly with the service or indirectly with it 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 can be used.

In this chapter, we will focus our attention on a direct communication style called *request-response*, in which a client sends a *request message* to the service, and the service replies back 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 format impacts 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 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 is leaner and more performant than a textual one at the expense of human readability.

When a client sends a request to a service, 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 and C#, can completely hide callbacks through language primitives such as async/await. These primitives make writing asynchronous code as straightforward as writing a synchronous one.

The most commonly used IPC technologies for request-response interactions are gRPC, REST, and GraphQL. Typically, internal APIs used for service-to-service 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 on REST. In

the rest of the chapter, we will walk through the process of creating a RESTful HTTP API.

5.1 HTTP

HTTP 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 5.2.

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 what the response's result is.
- The *headers* are key-value pairs with meta-information that describe the message.
- The message's *body* is a container for data.

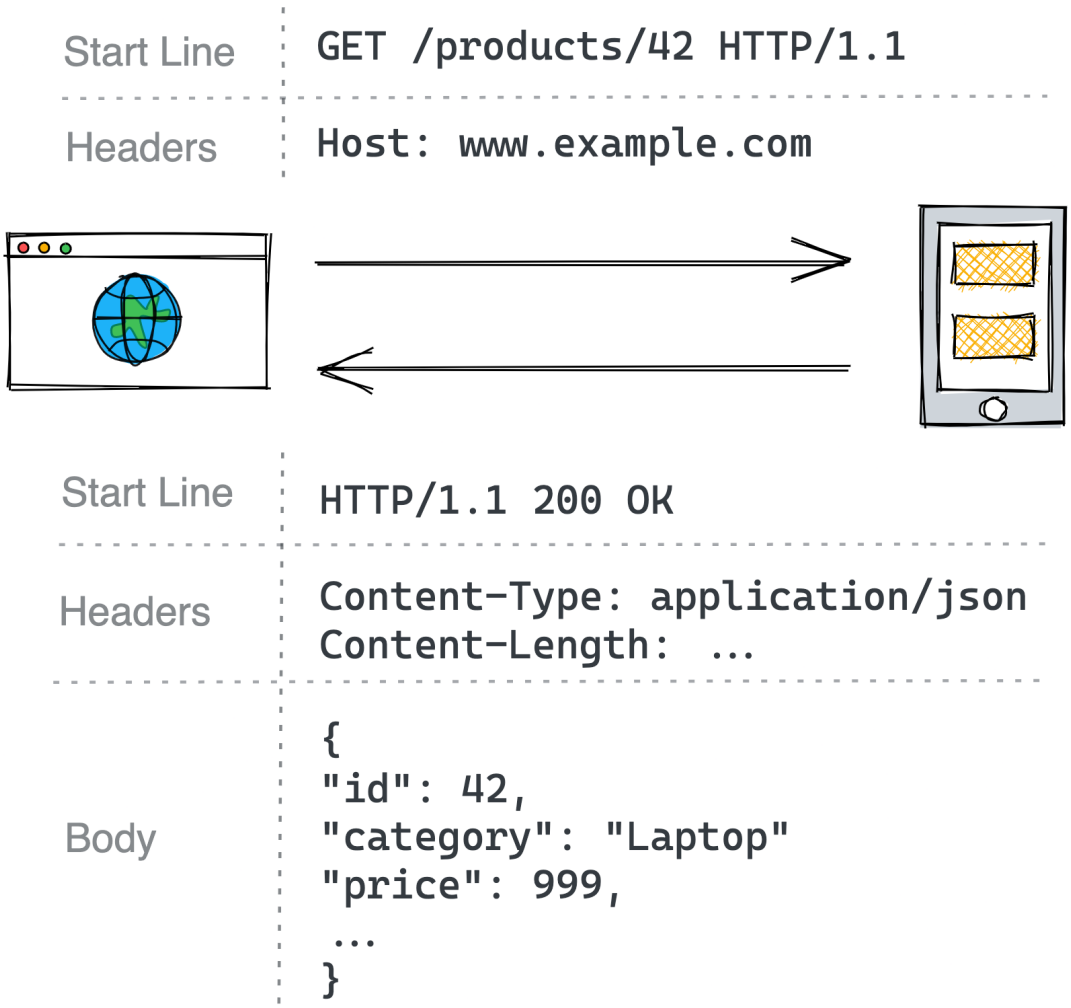


Figure 5.2: An example HTTP transaction between a browser and a web server.

HTTP is a stateless protocol, which means that everything needed by a server to process a request needs to be specified within the request itself, without context from previous requests. HTTP uses TCP for the reliability guarantees discussed in chapter 2. When it rides on top of TLS, it's also referred to as HTTPS. Needless to say, you should use HTTPS by default.

HTTP 1.1 keeps a connection to a server open by default to avoid creating a new one when the next transaction occurs. Unfortunately, a new request can't be issued until the response of the previous one has been received; 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 the time, which can be very inefficient.

Although HTTP 1.1 technically allows some type of requests to be pipelined, it has never been widely adopted due to its limitations. With HTTP 1.1, the typical way to improve the throughput of outgoing requests is by creating multiple connections. Although it comes with a price because connections consume resources like memory and sockets.

HTTP 2 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, which allows HTTP 2 to multiplex multiple concurrent request-response transactions 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 is the latest iteration of the HTTP standard, which is slowly being rolled out to browsers as I write this — it's based on UDP and implements its own transport protocol to address some of TCP's shortcomings.

Given that neither HTTP 2 nor HTTP 3 are ubiquitous yet, you still need to be familiar with HTTP 1.1, which is the standard the book uses going forward as its plain text format is easier to depict.

5.2 Resources

Suppose we are responsible for implementing a service to manage the product catalog of an e-commerce application. The service must allow users to browse the catalog and admins to create, update, or delete products. Sounds simple enough; the interface of the service could be defined like this:

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

External clients can't invoke interface methods directly, which is where the HTTP adapter comes in. It handles an HTTP request by invoking the methods defined in the service interface and converts their return values into HTTP responses. But to perform this mapping, we first need to understand how to model the API with HTTP in the first place.

An HTTP server hosts resources. A *resource* is an abstraction of information, like a document, an image, or a collection of other resources. It's identified by a URL, which describes the location of the resource 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 the way the request is handled by the service; 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.

HTTP gives us a lot of flexibility on how to design our API. Nothing forbids us from creating a resource name that looks like a remote procedure, like */getProducts*, which expects the additional parameters to be specified in the request's body, rather than in the query string. But if we were to do this, we could no longer cache the list of products by its URL. This is where REST comes in — it's a set of conventions and constraints for designing elegant and scalable HTTP APIs. In the rest of this chapter, we will use REST principles where it makes sense.

How should we model relationships? For example, a specific product is a resource that belongs to the collection of products, and that should ideally be reflected in its URL. Hence, the product with the unique identifier 42 could be identified with the relative URL */products/42*. The product could

also have a list of reviews associated with it, which we can model by appending the nested resource name, *reviews*, after the parent one, */products/42*, e.g., */products/42/reviews*. If we were to continue to add more nested resources, the API would become complex. As a rule of thumb, URLs should be kept simple, even if it means that the client might have to perform multiple requests to get the information it needs.

Now that we know how to refer to resources, let's see how to represent them on the wire when they are transmitted in the body of request and response messages. A resource can be represented in different ways; for example, a product can be represented either with an XML or a JSON document. JSON is typically used to represent non-binary resources in REST APIs:

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

When a client sends a request to a server to get a resource, it adds several headers to the message to describe its preferred representation. The server uses these headers to pick the most appropriate representation for the resource and decorates the response message with headers that describe it.

5.3 Request methods

HTTP requests can create, read, update, and delete (CRUD) resources by 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 URI of the new resource.

- *GET /products* — Retrieve a list of products. The query string can be used to filter, paginate, and sort the collection. Pagination should be used to return a limited number of resources per call to prevent denial of service attacks.
- *GET /products/42* — Retrieve product 42.
- *PUT /products/42* — Update product 42.
- *DELETE /products/42* — Delete product 42.

Request methods can be classified depending on whether they are safe and idempotent. A *safe* method should not have any visible side effects and can be safely 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.

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

The concept of idempotency is crucial and will come up repeatedly in the rest of the book, not just in the context of HTTP requests. An idempotent request makes it possible to safely retry requests that have succeeded, but for which the client never received a response; for example, because it crashed and restarted before receiving it.

5.4 Response status codes

After the service has received a request, it needs to send a response back to the client. The HTTP response contains a *status code* 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 continue to return the same error if it's retried, as the error is caused by an issue with the client, not the server. Because of that, it shouldn't be retried. These client errors are common:

- *400 (Bad Request)* — Validating the client-side input has failed.
- *401 (Unauthorized)* — The client isn't authorized to access a resource.
- *403 (Forbidden)* — The user is authenticated, but it's not allowed to access a 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 caused it to fail might be fixed by the time the retry is processed by the server. These are some typical server status codes:

- *500 (Internal Server Error)* — A generic server error.
- *502 (Bad Gateway)* — Indicates an invalid response from an upstream server.
- *503 (Service Unavailable)* — Indicates that the server can't currently serve the request, but might be able to in the future.

5.5 OpenAPI

Now that we have learned how to map the operations defined by our service's interface onto RESTful HTTP endpoints, we can formally define the API with an *interface definition language* (IDL), a language independent description of it. The IDL definition can be used to generate boilerplate code for the IPC adapter and client SDKs in your languages of choice.

The OpenAPI specification, which evolved from the Swagger project, is one of the most popular IDL for RESTful APIs based on HTTP. With it, we can formally describe our 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:

```yaml
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:
                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 spend time describing OpenAPI further as it's mostly an implementation detail, 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 for our languages of choice.

5.6 Evolution

APIs start out as beautifully-designed interfaces. Slowly, but surely, they will need to change over time to adapt to new use cases. The last thing you want to do when evolving your API is to introduce a breaking change that requires modifying all the clients in unison, some of which you might have no control over in the first place.

There are two types of changes that can break compatibility, one at the endpoint level and another at the message level. For example, if you were to change the */products* endpoint to */fancy-products*, it would obviously break clients that haven't been updated to support the new endpoint. The same goes when making a previously optional query parameter mandatory.

Changing the schema of request and 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 as the old deserialization logic would blow up in clients. Similar arguments can be made for messages represented with other serialization formats, like Protocol Buffers.

To support breaking changes, REST APIs should be versioned by either prefixing a version number in the URLs (e.g., */v1/products/*), using a custom header (e.g., *Accept-Version: v1*) or the *Accept* header with content negotiation (e.g., *Accept: application/vnd.example.v1+json*).

As a general rule of thumb, you should try to evolve your API in a backwards-compatible way unless you have a very good reason, in which case you need to be prepared to deal with the consequences. Backwards-compatible APIs tend to be not particularly elegant, but they are a necessary evil. There are tools that can compare the IDL specifications of your API and check for breaking changes; use them in your continuous integration pipelines.

(PART) Coordination

Introduction

So far, we have learned how we can get two processes to communicate reliably and securely with each other. We didn't go into all this trouble just for the sake of it, though. The end goal has always been to use multiple processes, and services, to build a distributed application that gives its clients the illusion they interact with a single node.

Although achieving a perfect illusion is not always possible or desirable, it's clear that some degree of coordination is needed to build a distributed application. In this part, we will explore the core distributed algorithms at the heart of large scale services.

Chapter 6 introduces formal models that encode our assumptions about the behavior of nodes, communication links, and timing; think of them as abstractions that allow us to reason about distributed systems by ignoring the complexity of the actual technologies used to implement them.

Chapter 7 describes how to detect that a remote process is unreachable. Since the network is unreliable and processes can crash at any time, a process trying to communicate with another could hang forever without failure detection.

Chapter 8 dives into the concept of time and order. In this chapter, we will first learn why agreeing on the time an event happened in a distributed system is much harder than it looks, and then propose 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 who can perform operations that others can't, like accessing a shared resource or coordinating other processes' actions.

Chapter 10 introduces one of the fundamental challenges in distributed systems, namely keeping replicated data in sync across multiple nodes. This

chapter explores why there is a tradeoff between consistency and availability and describes how the Raft replication algorithm works.

Chapter 11 dives into how to implement transactions that span data partitioned among multiple nodes or services. Transactions relieve you from a whole range of possible failure scenarios so that you can focus on the actual application logic rather than all possible things that can go wrong.

6 System models

To reason about distributed systems, we need to define precisely what can and can't happen. A *system model* encodes assumptions about the behavior of nodes, 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.

Let's start by introducing some models for communication links:

- The *fair-loss link* model assumes that messages may be lost and duplicated. 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 message's sender.

Even though these models are just abstractions of real communication links, they are useful to verify the correctness of algorithms. 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 does precisely that (and more), while TLS implements authentication (and more).

We can also model the different types of node failures we expect to happen:

- The *arbitrary-fault* model assumes that a node can deviate from its algorithm in arbitrary ways, leading to crashes or unexpected behavior due to bugs or malicious activity. The arbitrary fault model is also referred to as the "Byzantine" model for historical reasons. Interestingly, it can be theoretically proven that a system with

Byzantine nodes can tolerate up to $\frac{1}{3}$ of faulty nodes and still operate correctly.
- The *crash-recovery* model assumes that a node 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 node doesn't deviate from its algorithm, but if it crashes it never comes back online.

While it's possible to take an unreliable communication link and convert it into a more reliable one using a protocol (e.g., keep retransmitting lost messages), the equivalent isn't possible for nodes. Because of that, algorithms for different node models look very different from each other.

Byzantine node models are typically used to model safety-critical systems like airplane engine systems, nuclear power plants, financial systems, and other systems where a single entity doesn't fully control all the nodes[1]. These use cases are outside of the book's scope, and the algorithms presented will generally assume a crash-recovery model.

Finally, we can also model the timing assumptions:

- The *synchronous* model assumes that sending a message or executing an operation never takes over a certain amount of time. This is very unrealistic in the real world, where we know sending messages over the network can potentially take a very long time, and nodes can be stopped by, e.g., garbage collection cycles or page faults.
- The *asynchronous* model assumes that sending a message or executing an operation on a node 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.
- The *partially synchronous* model assumes that the system behaves synchronously most of the time, but occasionally it can regress to an asynchronous mode. This model is typically representative enough of practical systems.

In the rest of the book, we will generally assume a system model with fair-loss links, nodes with crash-recovery behavior, and partial synchrony. For the interested reader, "Introduction to Reliable and Secure Distributed Programming" is an excellent theoretical book that explores distributed algorithms for a variety of other system models not considered in this text.

But remember, models are just an abstraction of reality, and sometimes abstractions leak. As you read along, question the models' assumptions and try to imagine how algorithms that rely on them could break.

---

1. For example, digital cryptocurrencies such as Bitcoin implement algorithms that assume Byzantine nodes.↩

7 Failure detection

Several things can go wrong when a client sends a request to a server. In the happy path, the client sends a request and receives a response back. 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).

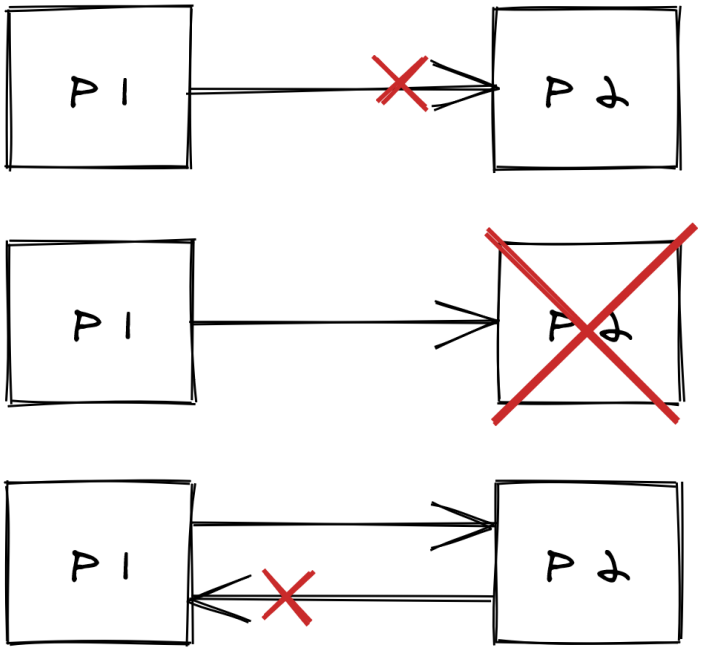


Figure 7.1: P1 can't tell whether P2 is slow, crashed or a message was delayed/dropped because of a network issue.

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 on whether the server is likely to be down or unreachable after some time has passed. To do that, 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 throws an error.

The tricky part is defining how long the amount of time that triggers this timeout should be. If it's too short and the server is reachable, the client will wrongly consider the server dead; if it's too long and the server is not reachable, the client will block waiting for a response. The bottom line is that it's not possible to build a perfect failure detector.

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

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 that doesn't happen, a timeout is triggered that marks the destination as dead. However, the process will keep regularly sending pings to it so that if and when it comes back online, it will reply to a ping and be marked as available again.

A *heartbeat* is a message that a process periodically sends to another to inform it that it's still up and running. If the destination doesn't receive a heartbeat within a specific time frame, it triggers a timeout and marks the process that missed the heartbeat as dead. If that process comes later back to life and starts sending out heartbeats, it will eventually be marked as available again.

Pings and heartbeats are typically used when specific processes frequently interact with each other, and an action needs to be taken as soon as one of them is no longer reachable. If that's not the case, detecting failures just at communication time is good enough.

8 Time

Time is an essential concept in any application, even more so in distributed ones. We have already encountered some use for it when discussing the network stack (e.g., DNS record TTL) and failure detection. Time also plays an important role in reconstructing the order of operations by logging their timestamps.

The flow of execution of a single-threaded application is easy to grasp since 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 and can be used to order their 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. Can you imagine designing a TCP-like protocol without using sequence numbers to order the packets? 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. The device you are using to read this book is likely using such a clock. It can run slightly faster or slower than others, depending on manufacturing differences and the external temperature. The rate at which a clock runs 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 measure time based on quantum-mechanical properties of

atoms and are significantly more expensive than quartz clocks and are accurate to 1 second in 3 million years.

The Network Time Protocol (NTP) is used to synchronize clocks. The challenge is to do so despite the unpredictable latencies introduced by the network. A NTP client estimates the clock skew by correcting the timestamp received by a NTP server with the estimated network latency. Armed with an estimate of the clock skew, the client can adjust its clock, causing it to jump forward or backward in time.

This creates a problem as measuring the elapsed time between two points in time becomes error-prone. For example, an operation that is executed after another could appear to have been executed before.

Luckily, most operating systems offer a different type of clock that is not affected by time jumps: the monotonic clock. A monotonic clock measures the number of seconds elapsed since an arbitrary point, like when the node started up, and can only move forward in time. A monotonic clock is useful to measure how much time elapsed between two timestamps on the same node, but timestamps of different nodes can't be compared with each other.

Since we don't have a way to synchronize wall-time clocks across processes perfectly, we can't depend on them for ordering operations. To solve this problem, we need to look at it from another angle. We know that two operations can't run concurrently in a single-threaded process as one must happen before the other. This happened-before relationship creates a causal bond between the two operations, as the one that happens first can change the state of the process and affect the operation that comes after it. We can use this intuition to build a different type of clock, one that isn't tied to the physical concept of time, but 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, which is 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* is a logical clock based on this idea. Every process in the system has its own local logical clock implemented with a numerical counter that follows specific rules:

- The counter is initialized with 0.
- The process increments its counter before executing an operation.
- When the process sends a message, it increments its counter and sends a copy of it in the message.
- When the process receives a message, its counter is updated to 1 plus the maximum of its current logical timestamp and the message's timestamp.

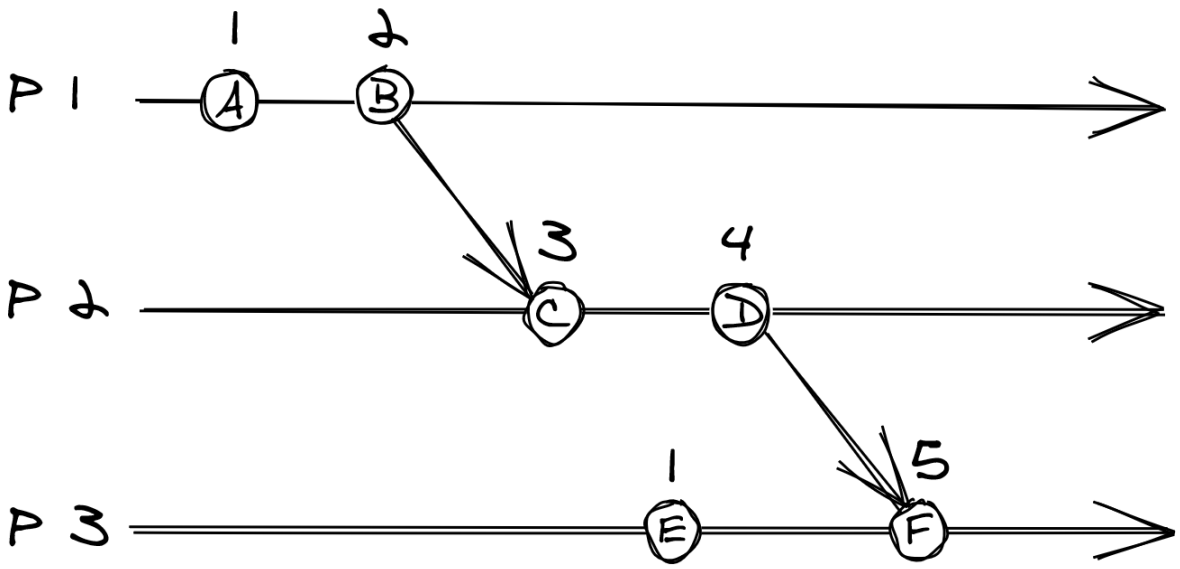


Figure 8.1: Three processes using Lamport clocks. For example, because D happened before F, D's logical timestamp is less than F's.

The Lamport clock assumes a crash-stop model, but a crash-recovery one can be supported by persisting the clock's state on disk, for example.

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.

You would think that the converse also applies — if the logical timestamp of operation O_3 is less than O_4, then O_3 happened-before O_4. But, that can't be guaranteed with Lamport timestamps. Going back to the example in Figure 8.1, operation E didn't happen-before C, even if their timestamps seem to imply it. To guarantee the converse relationship, we will have to use a different type of logical clock: the *vector clock*.

8.3 Vector clocks

A *vector clock* is a logical clock that guarantees that if two operations can be ordered by their logical timestamps, then one must have happened-before the other. A vector clock is implemented with an array of counters, one for each process in the system. And similarly to how Lamport clocks are used, each process has its own local copy of the clock.

For example, if the system is composed of 3 processes $P1$, $P2$, and $P3$, each process has a local vector clock implemented with an array[1] of 3 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 own counter in the array by one.

- When the process sends a message, it increments its own counter in the array by one 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 own counter in the array by one.

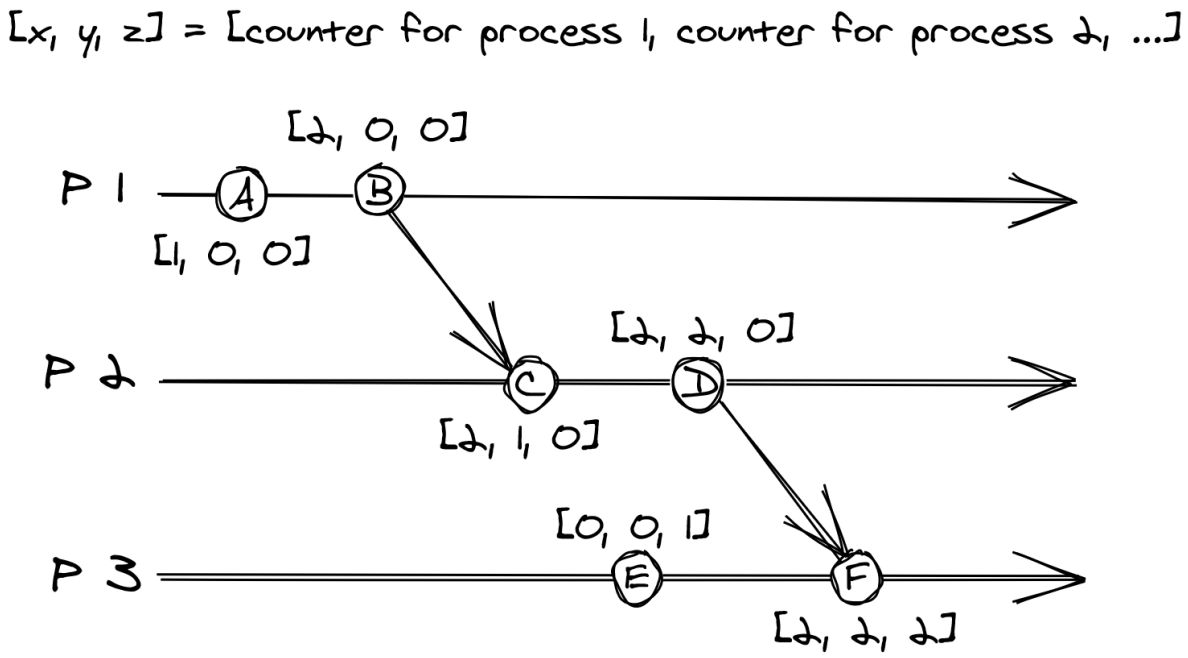


Figure 8.2: Each process has a vector clock represented with an array of three counters.

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

- every counter in T_1 is less than or equal 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. For example, operation E and C in Figure 8.2 can't be ordered, and therefore they are considered to be concurrent.

This discussion about logical clocks might feel quite abstract. Later in the book, we will encounter some practical applications of logical clocks. Once you learn to spot them, you will realize they are everywhere, as they can be disguised under different names. What's important to internalize at this point is that generally, you can't use physical clocks to derive accurately the order of events that happened on different processes[2] .

---

1. In actual implementations a dictionary is used rather than an array.↩

2. That said, sometimes physical clocks are good enough. For example, using physical clocks to timestamp logs is fine as they are mostly used for debugging purposes.↩

9 Leader election

Sometimes a single process in the system needs to have special powers, like being the only one that can access a shared resource or assign 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 crashes or becomes otherwise unavailable. When that happens, the remaining processes detect that the leader is no longer available and elect a new one.

A leader election algorithm needs to guarantee that there is at most one leader at any given time and that an election eventually completes. These two properties are also referred to as *safety* and *liveness*, respectively. This chapter explores how a specific algorithm, the *Raft* leader election algorithm, guarantees these properties.

9.1 Raft leader election

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

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

In Raft, time is divided into *election terms* of arbitrary length. An election term is represented with a logical clock, a numerical counter that can only increase over time. A term begins with a new election, during which one or more candidates attempt to become the leader. The algorithm guarantees that for any term there is at most one leader. 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 any heartbeat within a certain time period, a timeout fires and the leader is presumed dead. At that point, the follower starts a new election by incrementing the current election 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 the 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 sending out 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 the candidate's term, it accepts the new leader and returns to the follower state. 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 GC 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. When that happens, the candidate will eventually time out and start a new election. The election timeout is picked randomly from a fixed interval to reduce the likelihood of another split vote in the next election.

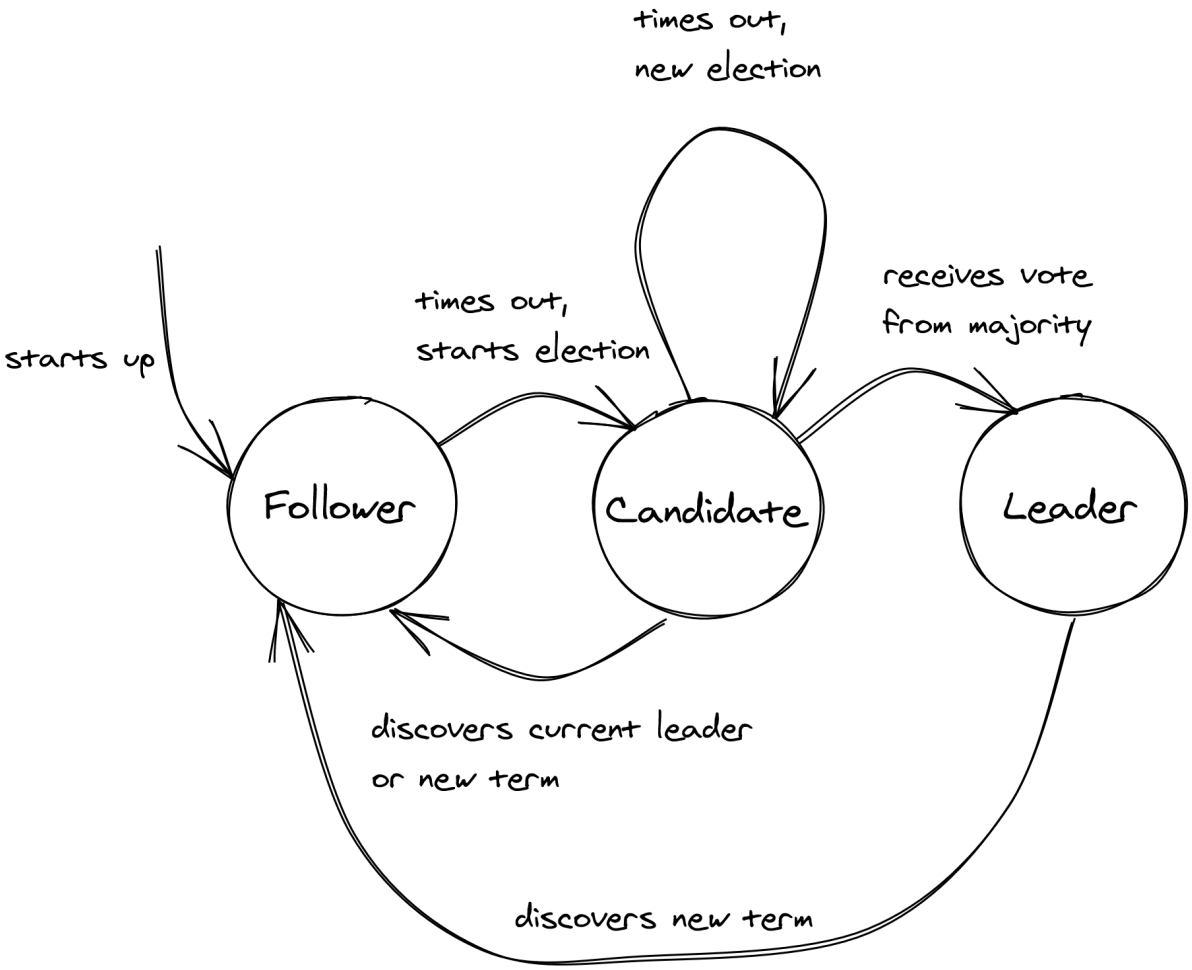


Figure 9.1: Raft's leader election algorithm represented as a state machine.

9.2 Practical considerations

There are many more leader election algorithms out there than the one presented here, but Raft's implementation is a modern take on the problem optimized for simplicity and understandability, which is why I chose it. That said, you will rarely need to implement leader election from scratch, as you can leverage linearizable key-value stores, like etcd or ZooKeeper, which offer abstractions that make it easy to implement leader election. The abstractions range from basic primitives like compare-and-swap to full-fledged distributed mutexes.

Ideally, the external store should at the very least offer an atomic compare-and-swap operation with an expiration time (TTL). The compare-and-swap operation updates the value of a key if and only if the value matches the expected one; the expiration time defines the time to live for a key, after which the key expires and is removed from the store if the lease hasn't been extended. The idea is that each competing process tries to acquire a lease by creating a new key with compare-and-swap using a specific TTL. 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 TTL expiry logic can also be implemented on the client-side, like this locking library 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 in your application. Unfortunately, that's not the case.

To see why, suppose there are multiple processes that need to update a file on a shared blob store, and you want to guarantee that only a single process at a time can do so to avoid race conditions. To achieve that, you decide to use a distributed mutex, a form of leader election. Each process tries to acquire the lock, and the one that does so successfully reads the file, updates it in memory, and writes it back to the store:

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

The problem here is that by the time the process writes the content to the store, it might no longer be the leader and a lot might have happened since it was elected. For example, the operating system might have preempted and stopped the process, and several seconds will have passed by the time it's running again. So how can the process ensure that it's still the leader

then? It could check one more time before writing to the store, but that doesn't eliminate the race condition, it just makes it less likely.

To avoid this issue, the data store downstream needs to verify that the request has been sent by the current leader. One way to do that is by using a fencing token. A *fencing token* is a number that increases every time that a distributed lock is acquired — in other words, it's a logical clock. When the leader writes to the store, it passes down the fencing token to it. The store remembers the value of the last token and accepts only writes with a greater value:

```
success, token = lock.acquire()
if success:
    try:
        content = store.read(blob_name)
        new_content = update(content)
        store.write(blob_name, new_content, token)
    except:
        lock.release()
```

This approach adds complexity as the downstream consumer, the blob store, needs to support fencing tokens. If it doesn't, you are out of luck, and you will have to design your system around the fact that occasionally there will be more than one leader. For example, if there are momentarily two leaders and they both perform the same idempotent operation, no harm is done.

Although having a leader can simplify the design of a system as it eliminates concurrency, it can become a scaling bottleneck if the number of operations performed by the leader increases to the point where it can no longer keep up. When that happens, you might be forced to re-design the whole system.

Also, having a leader introduces 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.

You 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.

Before considering the use of a leader, check whether there are other ways of achieving the desired functionality without it. For example, optimistic locking is one way to guarantee mutual exclusion at the cost of wasting some computing power. Or perhaps high availability is not a requirement for your application, in which case having just a single process that occasionally crashes and restarts is not a big deal.

As a rule of thumb, if you must use leader election, you have to minimize the work it performs and be prepared to occasionally have more than one leader if you can’t support fencing tokens end-to-end.

10 Replication

Data replication is a fundamental building block of distributed systems. One reason to replicate data is to increase availability. If some data is stored exclusively on a single node, and that node goes down, the data won't be accessible anymore. But if the data is replicated instead, clients can seamlessly switch to a replica. Another reason for replication is to increase scalability and performance; the more replicas there are, the more clients can access the data concurrently without hitting performance degradations.

Unfortunately replicating data is not simple, as it's challenging to keep replicas consistent with one another. In this chapter, we will explore Raft's replication algorithm, which is one of the algorithms that provide the strongest consistency guarantee possible — the guarantee that to the clients, the data appears to be located on a single node, even if it's actually replicated.

Raft is based on a technique known as *state machine replication*. The main idea behind it is that a single process, the leader, broadcasts the operations that change its state to other processes, the followers. If the followers execute the same sequence of operations as the leader, then the state of each follower will match the leader's. 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.

10.1 State machine replication

When the system starts up, a leader is elected using Raft's leader election algorithm, which we discussed in chapter 9. The leader is the only process that can make changes to the replicated state. It does so by storing the sequence of operations that alter the state into a local ordered log, which it then replicates to the followers; it's the replication of the log that allows the state to be replicated 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 index of the entry's position in the log;
- and the term number (the number in each box).

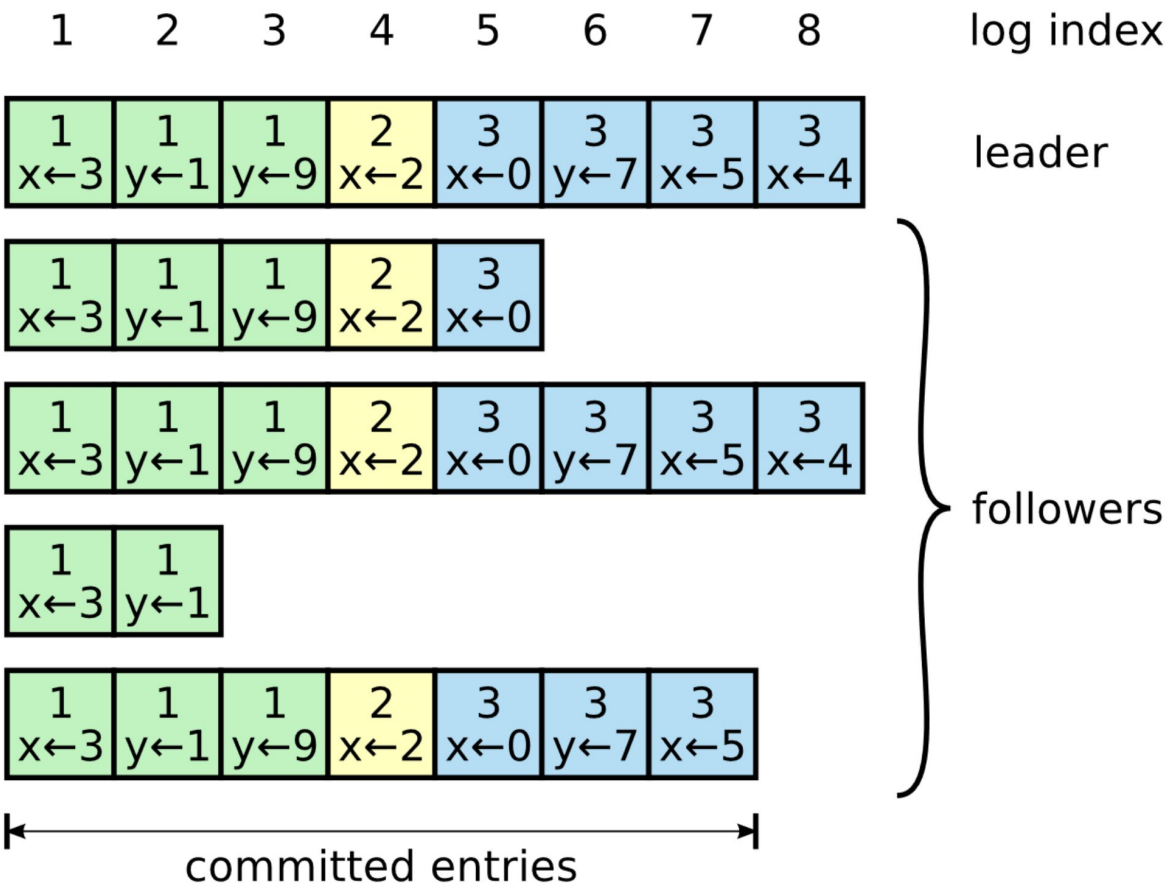


Figure 10.1: The leader's log is replicated to its followers. This figure appears in Raft's paper.

When the leader wants to apply an operation to its local state, it first appends a new log entry for the operation into 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 a so-called *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 log 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 committed log entry is considered to be durable and will eventually be executed by all available followers. 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 *only* for a majority of followers, it can make progress even if some processes 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 these 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 the processes to make progress, it's possible that when a leader fails, some processes are not up-to-date.

To avoid that an out-of-date process becomes 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 index and term of their last entries are compared. If the logs end with different terms, the log with the later 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.

What if a follower fails? 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 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.

So what happens when a follower that was temporarily unavailable comes back online? The resurrected follower 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 the same index and term number, it rejects the message, ensuring that an append to its log can't create a hole. It's as if the leader is sending a puzzle piece that the follower can't fit in its version of the puzzle.

When the *AppendEntries* request is rejected, the leader retries sending the message, this time including the last two log entries — this is why we referred to the request as *AppendEntries*, and not as *AppendEntry*. This dance continues until the follower finally accepts a list of log entries that can be appended to its log without creating a hole. Although the number of messages exchanged can be optimized, the idea behind it is the same: the follower waits for a list of puzzle pieces that perfectly fit its version of the puzzle.

10.2 Consensus

State machine replication can be used for much more than just replicating data since it's a solution to the consensus problem. *Consensus* is a fundamental problem studied in distributed systems research, which requires a set of processes to agree on a value in a fault-tolerant way 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.

Consensus has a large number of practical applications. For example, a set of processes agreeing which one should hold a lock or commit a transaction are consensus problems in disguise. As it turns out, deciding on a value can be solved with state machine replication. Hence, any problem that requires consensus can be solved with state machine replication too.

Typically, when you have a problem that requires consensus, the last thing you want to do is to solve it from scratch by implementing an algorithm like Raft. While it's important to understand what consensus is and how it can be solved, many good open-source projects implement state machine replication and expose simple APIs on top of it, like etcd and ZooKeeper.

10.3 Consistency models

Let's take a closer look at what happens when a client sends a request to a replicated store. 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 then needs to process it and finally send back a response to

the client. As shown in Figure 10.3, all these actions take time and are not instantaneous.

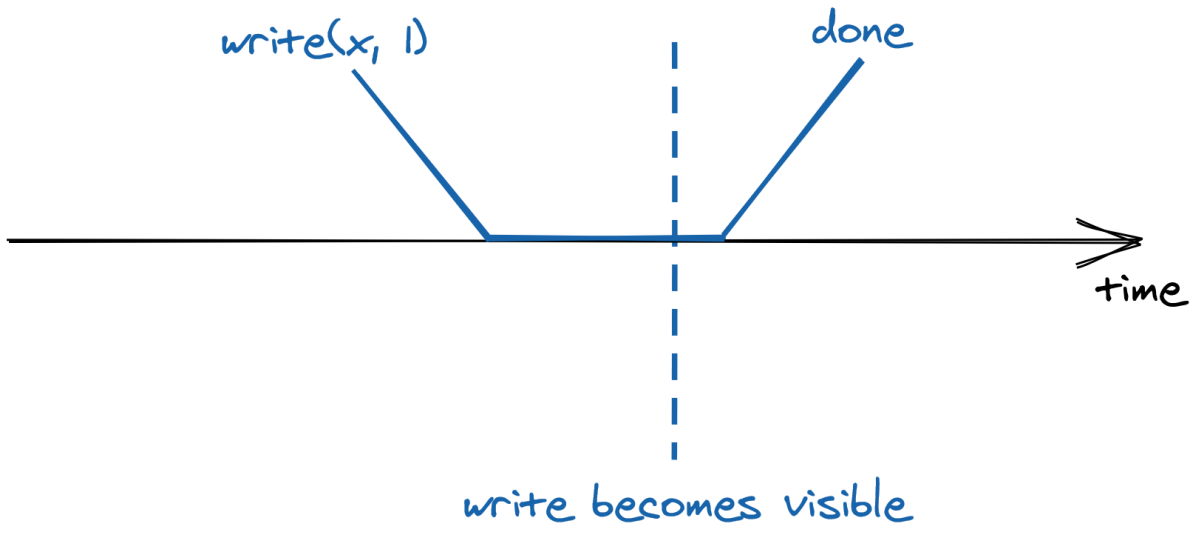


Figure 10.3: A write request can't execute instantaneously because it takes time to reach the leader and be executed.

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 what you are used to when writing single-threaded applications. If you assign 1 to x and read its value right after, you expect to find 1 in there, assuming there is no other thread writing to the same variable. But, once you start dealing with systems that replicate their state on multiple nodes for high availability and scalability, all bets are off. To understand why that's the case, we will explore different ways to implement reads in our replicated store.

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 were to go through the leader, the read throughput would be limited by that of a single process. But, if reads can be served by any follower 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 trade-off 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*, which formally define the possible views of the system's state observers can experience.

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 was 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, from their point of view there is a single copy of the data.

Because a request is not served instantaneously, and there is a single process serving it, the request executes somewhere between its invocation and completion time. Another way to think about it is that once a request completes, it's side-effects are visible to all observers as shown in Figure 10.4.

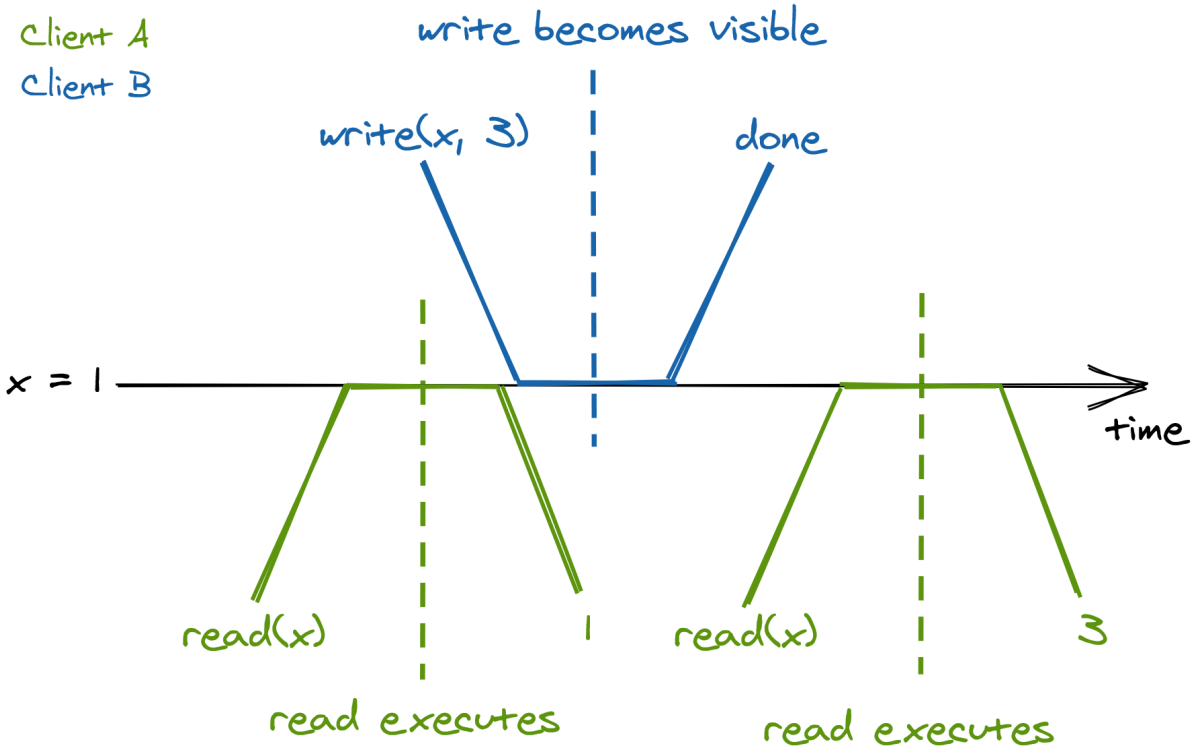


Figure 10.4: The side-effects of a strongly consistent operation are visible to all observers once it completes.

Since a request becomes visible to all other participants between its invocation and completion time, there is a real-time guarantee that must be enforced; this guarantee is formalized by a consistency model called *linearizability*, or strong consistency. Linearizability is the strongest consistency guarantee a system can provide for single-object requests.

What if the client sends a read request to the leader and by the time the request gets there, the server assumes it's the leader, but it actually was just deposed? If the ex-leader was to process the request, the system would no longer be strongly consistent. To guard against this case, the presumed leader first needs to contact a majority of the replicas to confirm whether it still is the leader. Only then it's allowed to execute the request and send back the response to the client. This 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 choke point, which limits 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 allow the followers to handle requests as well.

Even though a follower can lag behind the leader, it will always receive new updates in the same order as the leader. Suppose a 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 entirely in sync (see Figure 10.5).

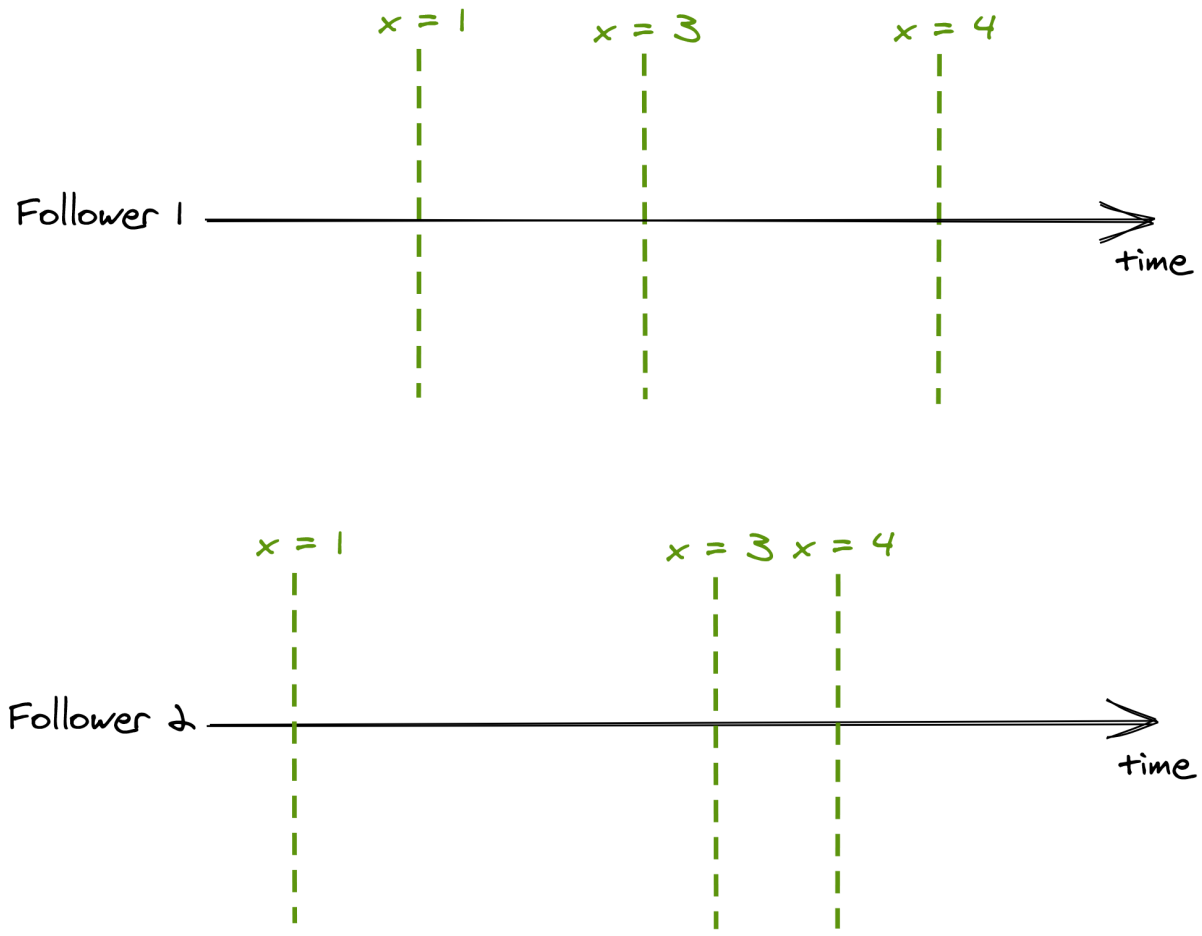


Figure 10.5: Although followers have a different view of the systems' state, they process updates in the same order.

The consistency model in which 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*. The lack of real-time guarantees is what differentiates sequential consistency from linearizability.

A producer/consumer system synchronized with a queue is an example of this model you might be familiar with; a producer process 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 — what if a follower goes down? We could increase the availability of the store by allowing a client to query any follower. But, this comes at a steep price in terms of consistency. Say there are two followers, 1 and 2, where follower 2 lags behind follower 1. If a client queries follower 1 and right after follower 2, it will see a state from the past, which can be very confusing. The only guarantee the client has is that eventually, all followers will converge to the final state if the 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 the one you are used to when writing single-threaded applications. Subtle bugs can creep up that are hard to debug and reproduce. Yet, in eventual consistency's defense, not all applications require linearizability. You need to make the conscious choice whether the guarantees offered by your data store, or lack thereof, satisfy your application's requirements.

An eventually consistent store is perfectly fine if you want to keep track of the number of users visiting your website, as it doesn't really matter if a read returns a number that is slightly out of date. But for a payment processor, you definitely want strong consistency.

10.3.4 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 followers to serve reads, sacrificing strong consistency;
- or guarantee strong consistency by failing reads that can't reach the leader.

This concept is expressed by the *CAP theorem*, 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.

Even though network partitions can happen, they are usually rare. But, there is a trade-off between consistency and latency in the absence of a network partition. The stronger the consistency guarantee is, the higher the latency of individual operations must be. This relationship is expressed by the *PACELC theorem*. It states that in case of network partitioning (P) in a distributed computer system, 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).

10.4 Practical considerations

To provide high availability and performance, off-the-shelf distributed data stores — sometimes referred to as *NoSQL* stores — come with counter-intuitive consistency guarantees. Others have knobs that allow you to choose whether you want better performance or stronger consistency guarantees, like Azure's Cosmos DB and Cassandra. Because of that, you need to know what the trade-offs are. With what you have learned here, you will be in a much better place to understand why the trade-offs are there in the first place and what they mean for your application.

11 Transactions

Transactions provide the illusion that a group of operations that modify some data has exclusive access to it and that either all operations complete successfully, or none does. You can typically leverage transactions to modify data owned by a single service, as it's likely to reside in a single data store that supports transactions. On the other hand, transactions that update data owned by different services, each with its own data store, are challenging to implement. This chapter will explore how to add transactions to your application when your data model is partitioned.

11.1 ACID

Consider a money transfer from one bank account to another. If the withdrawal succeeds, but the deposit doesn't, the funds need to be deposited back into the source account — money can't just disappear into thin air. In other words, the transfer needs to execute atomically; either both the withdrawal and the deposit succeed, or neither do. To achieve that, the withdrawal and deposit need to be wrapped in an inseparable unit: 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 they are rolled back as if they never happened.
- **C**onsistency guarantees that the application-level invariants, like a column that can't be null, must always be true. Confusingly, the "C" in ACID has nothing to do with the consistency models we talked about so far, and according to Joe Hellerstein, the "C" was tossed in to make the acronym work. Therefore, we will safely ignore this property in the rest of this chapter.

- **I**solation guarantees that the concurrent execution of transactions doesn't cause any race conditions.
- **D**urability guarantees that once the data store commits the transaction, the changes are persisted on durable storage. The use of a write-ahead log (WAL) is the standard method used to ensure durability. When using a WAL, the data store can update its state only after log entries describing the changes have been flushed to permanent storage. Most of the time, the database doesn't read from this log at all. But if the database crashes, the log can be used to recover its prior state.

Transactions relieve you from a whole range of possible failure scenarios so that you can focus on the actual application logic rather than all possible things that can go wrong. This chapter explores how distributed transactions differ from ACID transactions and how you can implement them in your systems. We will focus our attention mainly on atomicity and isolation.

11.2 Isolation

A set of concurrently running transactions that access 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 been 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 a committed transaction updated the value between the two reads.
- A *phantom read* happens when a transaction reads a set of objects matching a specific condition, while another transaction adds, updates, or deletes an object matching the same condition. For example, if one transaction is summing all employees' salaries while another deletes some employee records simultaneously, the sum of the salaries will be wrong 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.

Transactions can have different types of isolation levels that are defined based on the type of race conditions they forbid, as shown in Figure 11.1.

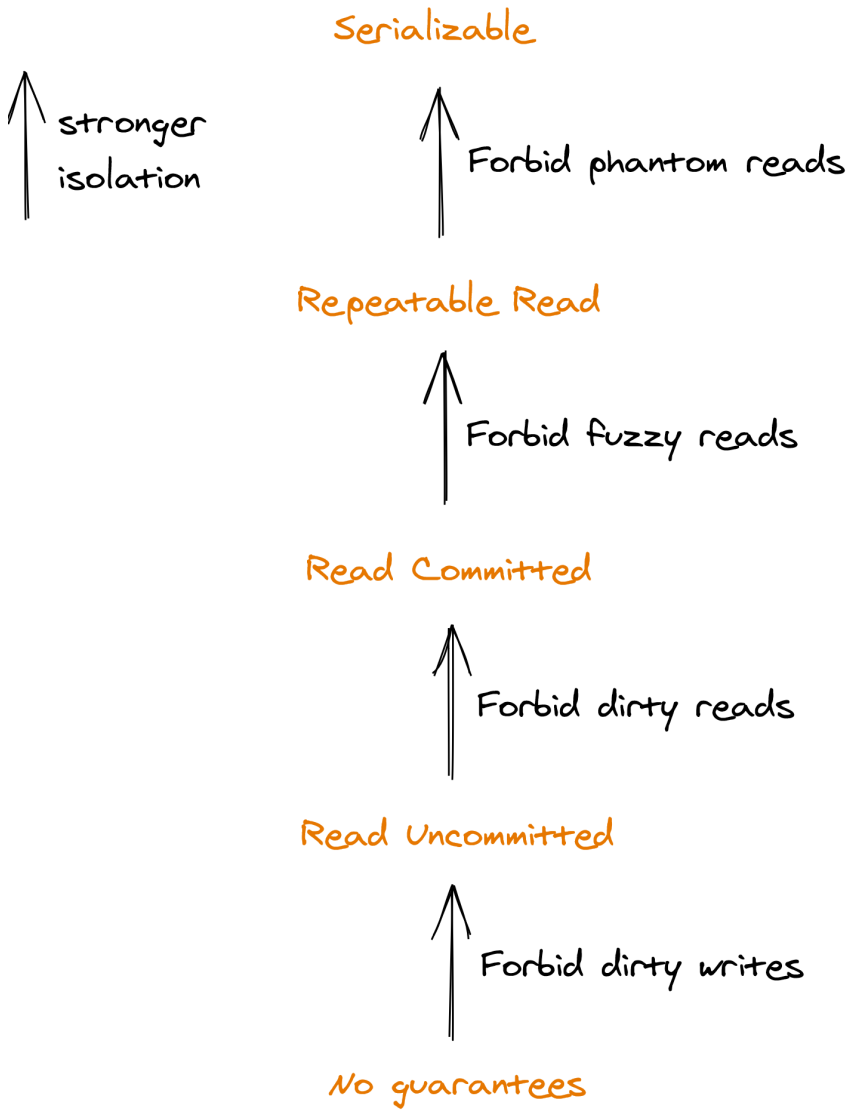


Figure 11.1: Isolation levels define which race conditions they forbid.

Serializability is the only isolation level that guards against all possible race conditions. It guarantees that the side effects of executing a set of transactions appear to be the same as if they had executed sequentially, one after the other. But, we still have a problem — there are many possible orders that the transactions can appear to be executed in, as serializability doesn't say anything about which one to pick.

Suppose we have two transactions A and B, and transaction B completes 5 minutes after transaction A. A system that guarantees serializability can reorder them so that B's changes are applied before A's. 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 that linearizability provides 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. As a result, there are many different isolation levels that are simpler to implement and also perform better. Your application might not need serializability, but you need to consciously decide which isolation level to use and understand its implications, or your data store will silently make the decision for you; for example, PostgreSQL's default isolation is read committed. When in doubt, choose strict serializability.

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

Now that we know what serializability is, let's look at how it can be implemented and why it's so expensive in terms of performance. Serializability can be achieved either with a pessimistic or an optimistic concurrency control mechanism.

11.2.1 Concurrency control

Pessimistic concurrency control uses locks to block other transactions from accessing a data item. The most popular pessimistic protocol is *two-phase locking* (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 data item in read-only mode, but it blocks transactions trying to acquire a write lock. The latter can be held only by a single transaction and blocks anyone else trying to acquire either a read or write lock on the data item.

There are two phases in 2PL, an expanding and a shrinking one. In the expanding phase, the transaction is allowed only to acquire locks, but not to release them. In the shrinking phase, the transaction is permitted only to release locks, but not to acquire them. If these rules are obeyed, it can be formally proven that the protocol guarantees serializability.

The *optimistic* approach to concurrency control doesn't block, as it checks for conflicts only at the very end of a transaction. If a conflict is detected, the transaction is aborted or restarted from the beginning. Generally, optimistic concurrency control is implemented with *multi-version concurrency control* (MVCC). With MVCC, the data store keeps multiple versions of a data item. Read-only transactions aren't blocked by other transactions, as they can keep reading the version of the data that was committed at the time the transaction started. But, a transaction that writes to the store is aborted or restarted when a conflict is detected. While MVCC per se doesn't guarantee serializability, there are variations of it that do, like Serializable Snapshot Isolation (SSI).

Optimistic concurrency makes sense when you have read-heavy workloads that only occasionally perform writes, as reads don't need to take any locks. For write-heavy loads, a pessimistic protocol is more efficient as it avoids retrying the same transactions repeatedly.

I have deliberately not spent much time describing 2PL and MVCC, as it's unlikely you will have to implement them in your systems. But, the commercial data stores your systems depend on use one or the other

technique to isolate transactions, so you must have a basic grasp of the tradeoffs.

11.3 Atomicity

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. How should we go about guaranteeing atomicity across the two accounts? We can't just run two separate transactions to respectively withdraw and deposit the funds — if the second transaction fails, then the system is left in an inconsistent state. We need atomicity: the guarantee that either both transactions succeed and their changes are committed, or that they fail without any side effects.

11.3.1 Two-phase commit

Two-phase commit (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*. Generally, the client application that initiates the transaction acts as the coordinator for the protocol.

When a coordinator wants to commit a transaction to the participants, it sends a prepare request asking the participants whether they are prepared to commit the transaction (see Figure 11.2). If all participants reply that they are ready to commit, the coordinator sends out a commit message 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.

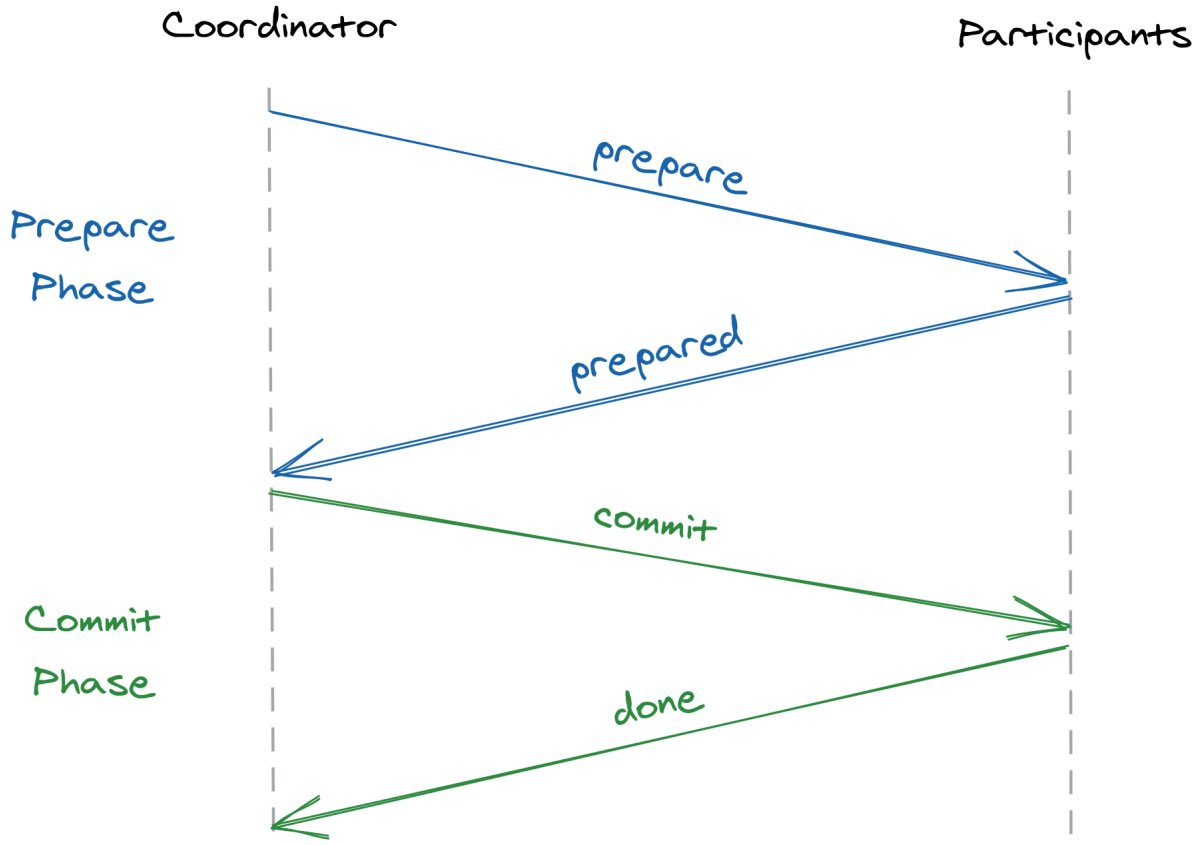


Figure 11.2: The two-phase commit protocol consists of a prepare and a commit phase.

There are two points of non-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 if the coordinator crashes, the participant is completely blocked.

The other point of non-return is when the leader 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 through that the transaction is 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. It's slow, as it requires multiple round trips to complete a transaction and blocks when there is a failure. 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 PostgreSQL and Cassandra and expect them to play ball with each other.

If we are willing to increase complexity, there is a way to make the protocol more resilient to failures. 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 than consensus.

Yet, an off-the-shelf consensus implementation can still be used to make 2PC more robust to failures. For example, replicating the coordinator with a consensus algorithm like Raft makes 2PC resilient to coordinator failures. Similarly, the participants could also be replicated.

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 as 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 mostly changed as distributed data stores have continued to add features that only traditional databases offered, and ACID transactions have become the norm rather than the exception. For example, Google's Spanner implements transactions across partitions using a combination of 2PC and state machine replication.

11.4 Asynchronous transactions

2PC is a synchronous blocking protocol; if any of the participants isn't available, the transaction can't make any progress, and the application blocks completely. The assumption is that the coordinator and the participants are available and that the duration of the transaction is short-

lived. Because of its blocking nature, 2PC is generally combined with a blocking concurrency control mechanism, like 2PL, to provide isolation.

But, some types of transactions can take hours to execute, in which case blocking just isn't an option. And some transactions don't need isolation in the first place. Suppose we were to drop the isolation requirement and the assumption that the transactions are short-lived. Can we come up with an asynchronous non-blocking solution that still provides atomicity?

11.4.1 Log-based transactions

A typical pattern in modern applications is replicating the same data in different data stores tailored to different use cases, like search or analytics. Suppose we own a product catalog service backed by a relational database, and we decided to offer an advanced full-text search capability in its API. Although some relational databases offer basic full-text search functionality, a dedicated database such as Elasticsearch is required for more advanced use cases.

To integrate with the search index, the catalog service needs to update both the relational database and the search index when a new product is added or an existing product is modified or deleted. The service could just update the relational database first, and then the search index; but if the service crashes before updating the search index, the system would be left in an inconsistent state. As you can guess by now, 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 2PC standard, the search index doesn't, which means we would have to implement that functionality from scratch. We also don't want the catalog service to block if the search index is temporarily unavailable. Although we want the two data stores to be in sync, we can accept some temporary inconsistencies. In other words, eventual consistency is acceptable for our use case.

To solve this problem, let's introduce an message log in our application. A log is an append-only, totally ordered sequence of messages, in which each

message is assigned a unique sequential index. Messages are appended at the end of the log, and consumers read from it in order. Kafka and Azure Event Hubs are two popular implementations of logs.

Now, when the catalog service receives a request from a client to create a new product, rather than writing to the relational database, or the search index, it appends a product creation message to the message log. The append acts as the atomic commit step for the transaction. The relational database and the search index are asynchronous consumers of the message log, reading entries in the same order as they were appended and updating their state at their own pace (see Figure 11.3). Because the message log is ordered, it guarantees that the consumers see the entries in the same order.

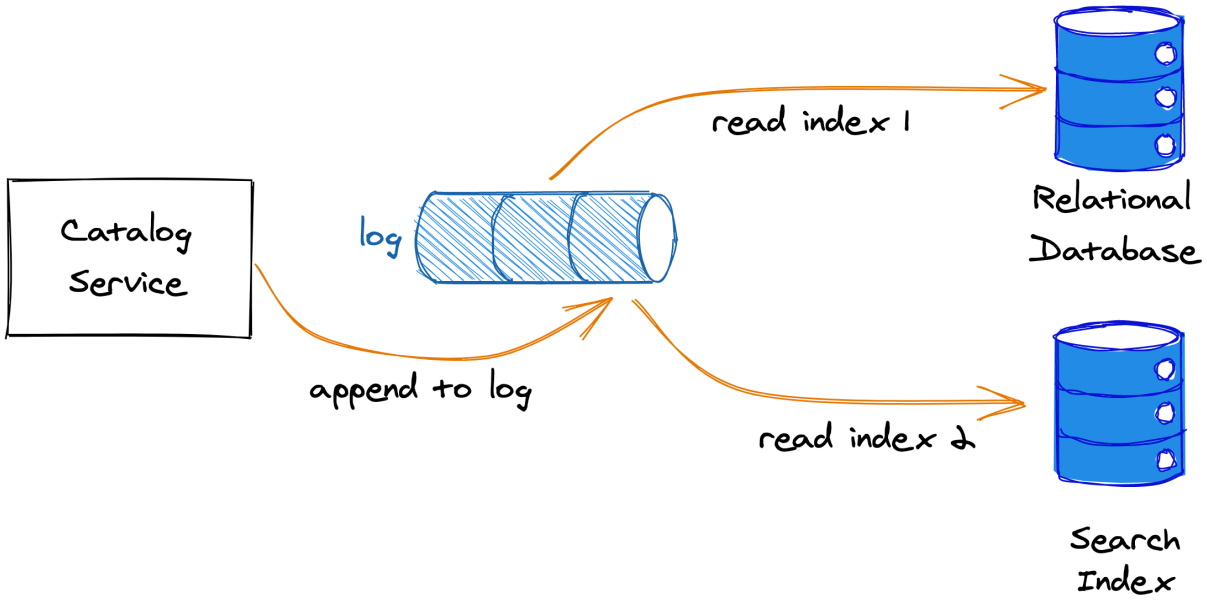


Figure 11.3: The producer appends entries at the end of the log, while the consumers read the entries at their own pace.

The consumers periodically checkpoint the index of the last message they processed. If a consumer crashes and comes back online after some time, it reads the last checkpoint and resumes reading messages from where it left off. Doing so ensures there is no data loss even if the consumer was offline for some time.

But, there is a problem as the consumer can potentially read the same message multiple times. For example, the consumer could process a message and crash before checkpointing its state. When it comes back online, it will eventually re-read the same message. Therefore, messages need to be idempotent so that no matter how many times they are read, the effect should be the same as if they had been processed only once. One way to do that is to decorate each message with a unique ID and ignore messages with duplicate IDs at read time.

We have already encountered the log abstraction in chapter 10 when discussing state machine replication. If you squint a little, you will see that what we have just implemented here is a form of state machine replication, where the state is represented by all products in the catalog, and the replication happens across the relational database and the search index.

Message logs are part of a more general communication interaction style referred to as *messaging*. In this model, the sender and the receiver don't communicate directly with each other; they exchange messages through a channel that acts as a broker. The sender sends messages to the channel, and on the other side, the receiver reads messages from it.

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.

A message has a well-defined format, consisting of a header and a body. The message header contains metadata, such as a unique message ID, while its 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 in the sender.

Services use inbound adapters to receive messages from messaging channels, which are part of their API surface, and outbound adapters to send messages, as shown in Figure 11.4. The log abstraction we have used earlier is just one form of messaging channel. Later in the book, we will

encounter other types of channels, like queues, that don't guarantee any ordering of the messages.

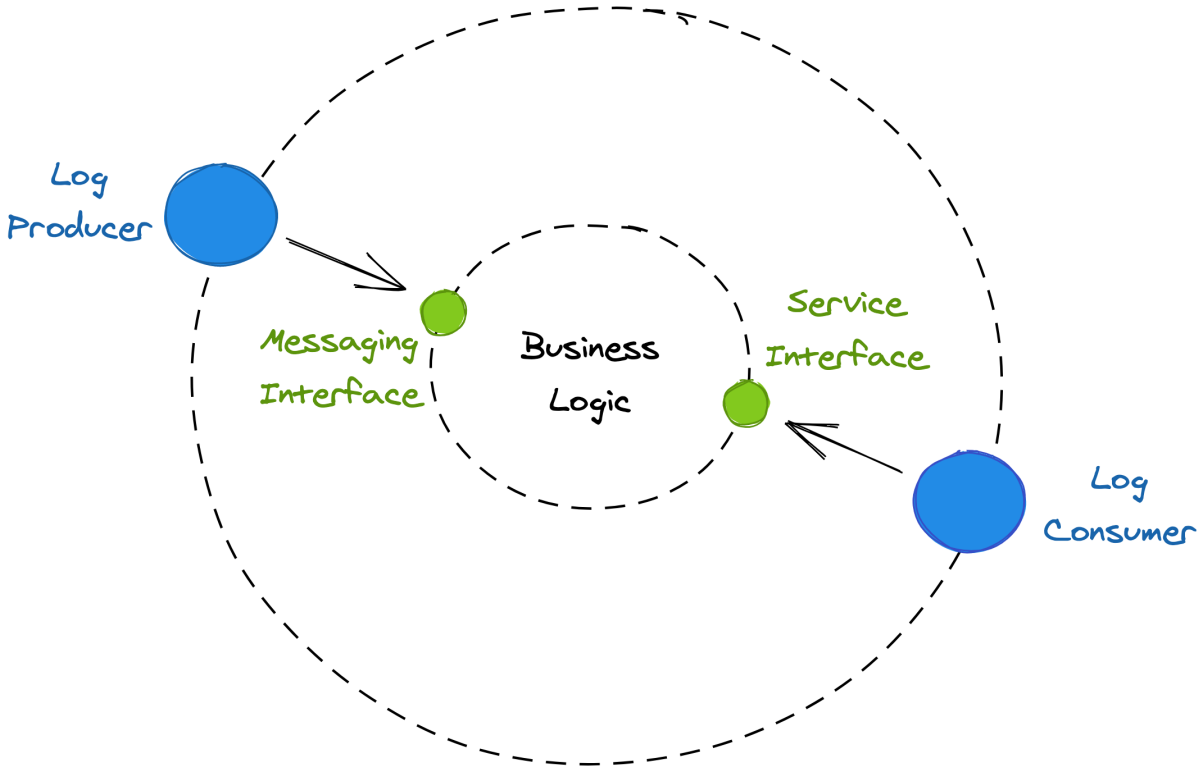


Figure 11.4: Inbound messaging adapters are part of a service's API surface.

11.4.2 Sagas

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 requests. 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. Since appending a single message to a log is no longer sufficient to commit the transaction, we can't use the simple log-oriented solution presented earlier.

The *Saga* 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, that 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. A Saga can be implemented with an orchestrator, the transaction's coordinator, that manages the execution of the local transactions across the processes involved, 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* depicted in Figure 11.5:

1. The coordinator initiates the transaction by sending a booking request to the flight service (T_1). 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 to the hotel service (T_2). 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 to the flight service to cancel the flight it previously booked (C_1). 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 temporarily unavailable ones. As the transaction requires multiple steps to succeed, and the coordinator 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 database 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.

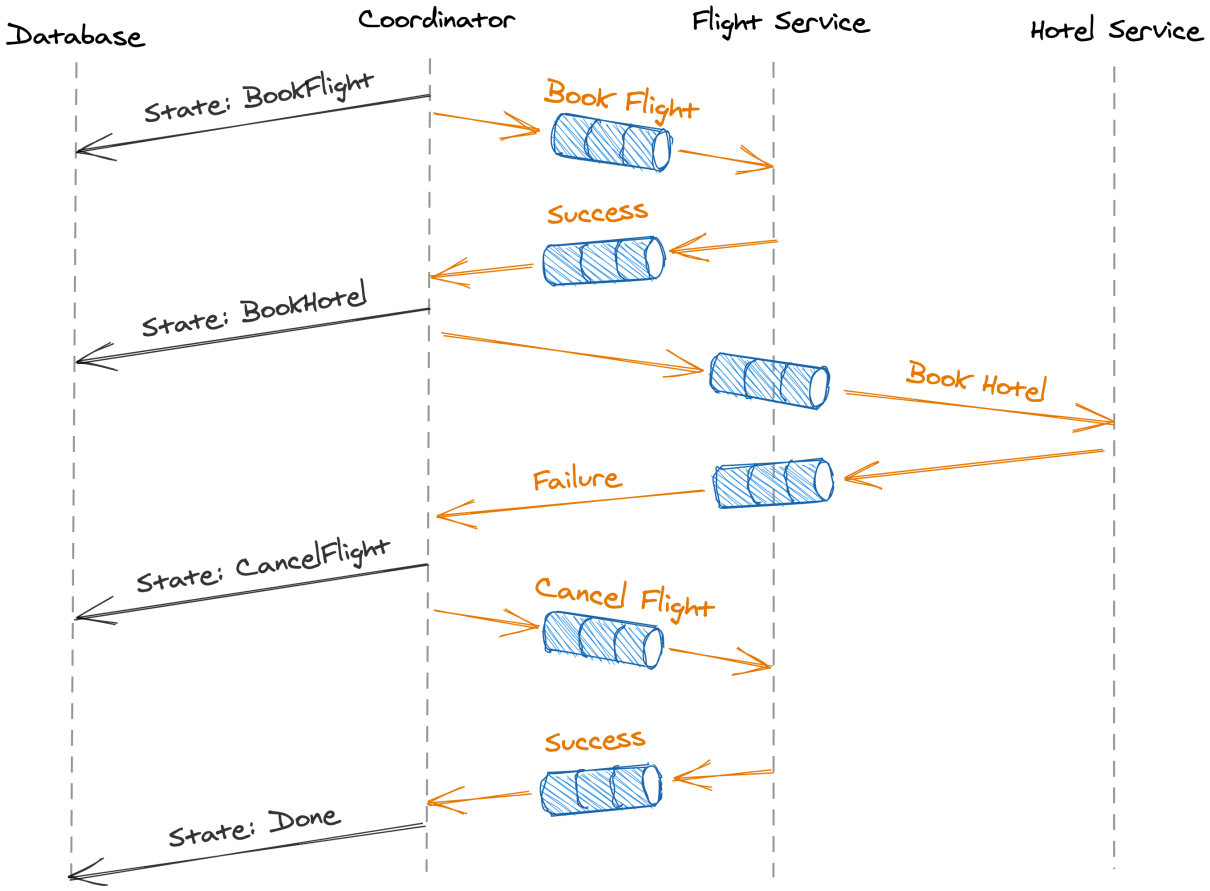


Figure 11.5: A workflow implementing an asynchronous transaction.

There is a caveat, though; if the coordinator crashes after sending a request but before backing up its state, it will have to re-send the request 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. Serverless cloud compute services such as AWS Step Functions or Azure Durable Functions make it easy to create fully-managed workflows.

11.4.3 Isolation

We started our journey into asynchronous transactions as a way to design around the blocking nature of 2PC. To get here, 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 with the use of *semantic locks*. The idea is that any data the Saga modifies is marked with a *dirty flag*. This flag is only cleared at the end of the transaction when it completes. Another transaction trying to access a dirty record can either fail and roll back its changes, or block until the dirty flag is cleared. The latter approach can introduce deadlocks, though, which requires a strategy to mitigate them.

(PART) Scalability

Introduction

Now that we understand how to coordinate processes, we are ready to dive into one of the main use cases for building distributed systems: scalability.

A scalable application can increase its capacity as its load increases. The simplest way to do that is by *scaling up* and running the application on more expensive hardware, but that only brings you so far since the application will eventually reach a performance ceiling.

The alternative to scaling up is *scaling out* by distributing the load over multiple nodes. This part explores 3 categories — or dimensions — of scalability patterns: *functional decomposition*, *partitioning*, and *duplication*. The beauty of these dimensions is that they are independent of each other and can be combined within the same application.

Functional decomposition

Functional decomposition is the process of taking an application and breaking it down into individual parts. Think of the last time you wrote some code; you most likely decomposed it into functions, classes, and modules. The same idea can be taken further by decomposing an application into separate services, each with its own well-defined responsibility.

Section 12.1 discusses the advantages and pitfalls of splitting an application into a set of independently deployable services.

Section 12.2 describes how external clients can communicate with an application after it has been decomposed into services using an API gateway. The gateway acts as the proxy for the application by routing, composing, and translating requests.

Section 12.3 discusses how to decouple an API's read path from its write path so that their respective implementations can use different technologies that fit their specific use cases.

Section 12.4 dives into asynchronous messaging channels that decouple producers on one end of a channel from consumers on the other end. Thanks to channels, communication between two parties is possible even if the destination is temporarily not available. Messaging provides several other benefits, which we will explore in this section, along with best practices and pitfalls you can run into.

Partitioning

When a dataset no longer fits on a single node, it needs to be partitioned across multiple nodes. Partitioning is a general technique that can be used in a variety of circumstances, like sharding TCP connections across backends in a load balancer.

We will explore different sharding strategies in section 13.1, such as range and hash partitioning. Then, in section 13.2, we will discuss how to rebalance partitions either statically or dynamically.

Duplication

The easiest way to add more capacity to a service is to create more instances of it and have some way of routing, or balancing, requests to them. This can be a fast and cheap way to scale out a stateless service, as long as you have considered the impact on the service's dependencies. Scaling out a stateful service is significantly more challenging as some form of coordination is required.

Section 14.1 introduces the concept of load balancing requests across nodes and its implementation using commodity machines. We will start with DNS load balancing and then dive into the implementation of load balancers that operate at the transport and application layer of the network stack. Finally, we will discuss geo load balancing that allows clients to communicate with the geographically closest datacenter.

Section 14.2 describes how to replicate data across nodes and keep it in sync. Although we have already discussed one way of doing that with Raft in chapter 10, in this section, we will take a broader look at the topic and

explore different approaches with varying trade-offs (single-leader, multi-leader, and leaderless).

Section 14.3 discusses the benefits and pitfalls of caching. We will start by discussing in-process caches first, which are easy to implement but have several pitfalls. Finally, we will look at the pros and cons of external caches.

12 Functional decomposition

12.1 Microservices

An application typically starts its life as a monolith. Take a modern backend of a single-page JavaScript application (SPA), for example. It might start out as a single stateless web service that exposes a RESTful HTTP API and uses a relational database as a backing store. The service is likely to be composed of a number of components or libraries that implement different business capabilities, as shown in Figure 12.1.

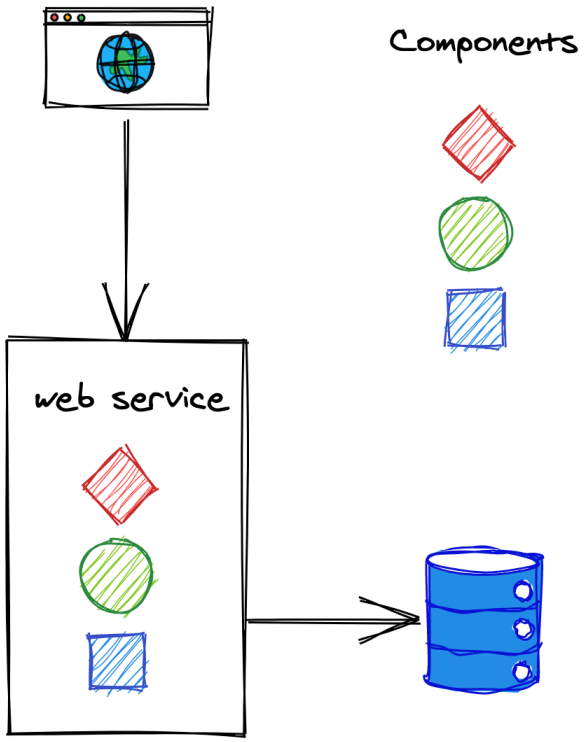


Figure 12.1: A monolithic backend composed of multiple components.

As the number of feature teams contributing to the same codebase increases, the components become increasingly coupled over time. This leads the teams to step on each other's toes more and more frequently, decreasing their productivity.

The codebase becomes complex enough that nobody fully understands every part of it, and implementing new features or fixing bugs becomes time-consuming. Even if the backend is componentized into different libraries owned by different teams, a change to a library requires the service to be redeployed. And if a change introduces a bug like a memory leak, the entire service can potentially be affected by it. Additionally, rolling back a faulty build affects the velocity of all teams, not just the one that introduced the bug.

One way to mitigate the growing pains of a *monolithic* backend is to split it into a set of independently deployable services that communicate via APIs, as shown in Figure 12.2. 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.

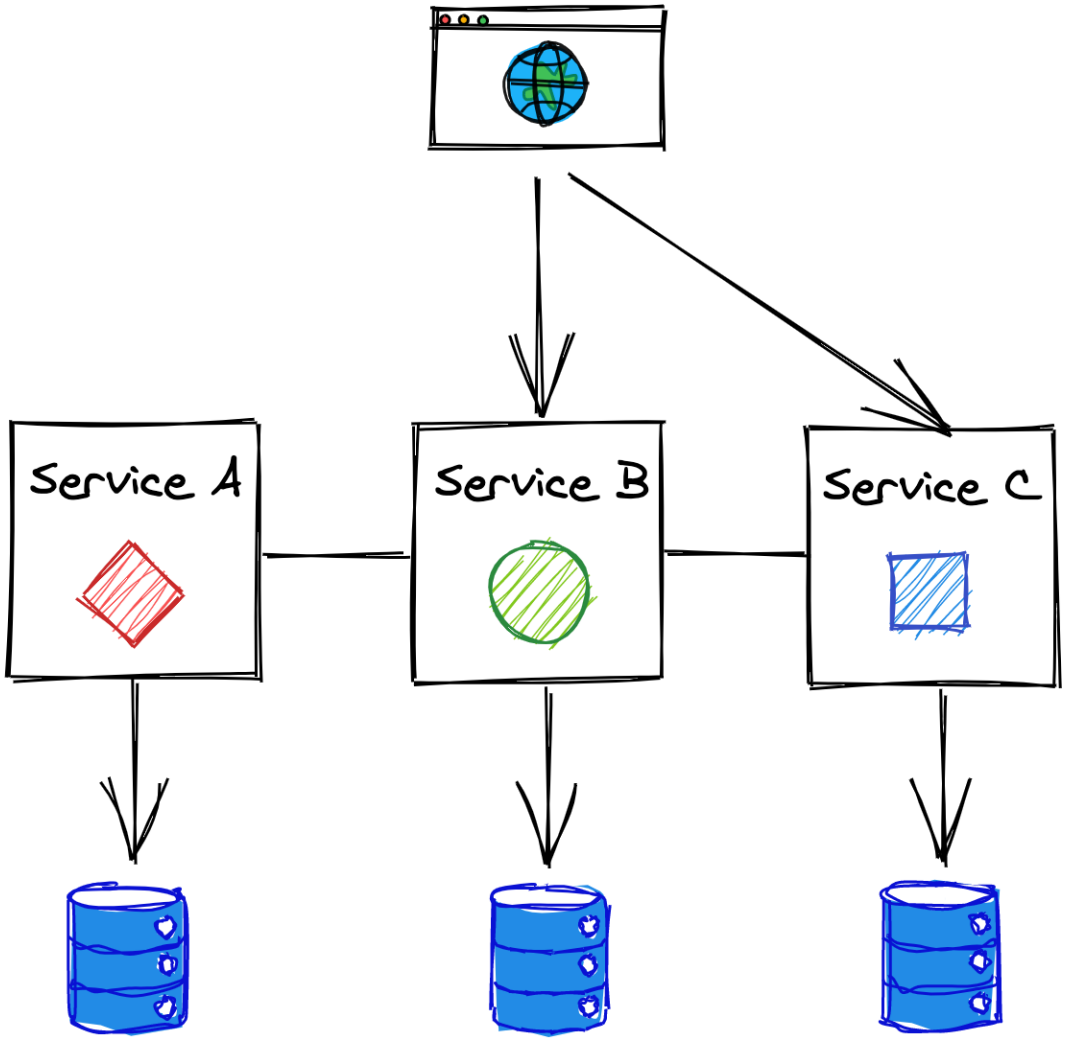


Figure 12.2: A backend split into independently deployable services that communicate via APIs.

This architectural style is also referred to as the *microservice architecture*. The term *micro* can be misleading, though — there doesn't have to be anything micro about the services. In fact, I would argue that if a service doesn't do much, it just creates more operational overhead than benefits. A more appropriate name for this architecture is service-oriented architecture, but unfortunately, that name comes with some old baggage as well. Perhaps in 10 years, we will call the same concept with yet another name, but for now we will have to stick to microservices.

12.1.1 Benefits

Breaking down the backend by business capabilities into a set of services with well-defined boundaries allows each service to be developed and operated by a single small team. Smaller teams can increase the application's development speed for a variety of reasons:

- They are more effective as the communication overhead grows quadratically with the team's size.
- Since each team dictates its own release schedule and has complete control over its codebase, less cross-team communication is required, and therefore decisions can be taken in less time.
- The codebase of a service is smaller and easier to digest by its developers, reducing the time it takes to ramp up new hires. Also, a smaller codebase doesn't slow down IDEs as much, which makes developers more productive.
- The boundaries between services are much stronger than the boundaries between components in the same process. Because of that, when a developer needs to change a part of the backend, they only need to understand a small part of the whole.
- Each service can be scaled independently and adopt a different technology stack based on its own needs. The consumers of the APIs don't care how the functionality is implemented after all. This makes it easy to experiment and evaluate new technologies without affecting other parts of the system.
- Each microservice can have its own independent data model and data store(s) that best fit its use-cases, allowing developers to change its schema without affecting other services.

12.1.2 Costs

The microservice architecture adds more moving parts to the overall system, and this doesn't come for free. The cost of fully embracing microservices is only worth paying if it can be amortized across dozens of development teams.

Development experience

Nothing forbids the use of different languages, libraries, and datastores in each microservice, but doing so transforms the application into an unmaintainable mess. For example, it becomes more challenging for a developer to move from one team to another if the software stack is completely different. 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 that a certain degree of standardization is needed. 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.

Resource provisioning

To support a large number of independent services, it should be simple to spin up 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 be able to pull this off, you will need a fair amount of automation.

Communication

Remote calls are expensive and come with all the caveats we discussed earlier in the book. You will need defense mechanisms to protect against failures and leverage asynchrony and batching to mitigate the performance hit of communicating across the network. All of this increases the system's complexity.

Much of what is described in this book is about dealing with this complexity, and as it should be clear by now, it doesn't come cheap. That being said, even a monolith doesn't live in isolation since it's being accessed by remote clients, and it's likely to use third-party APIs as well. So eventually, these issues need to be solved there as well, albeit on a smaller scale.

Continuous integration, delivery, and deployment

Continuous integration ensures that code changes are merged into the main branch after an automated build and test suites have run. Once a code change has been merged, it should be automatically published and deployed to a production-like environment, where a battery of integration and end-to-end tests run to ensure that the service doesn't break any dependencies or use cases.

While testing individual microservices is not more challenging than testing a monolith, testing the integration of all the microservices is an order of magnitude harder. Very subtle and unexpected behavior can emerge when individual services interact with each other.

Operations

Unlike with a monolith, it's much more expensive to staff each team responsible for a service with its own operations team. As a result, the team that develops a service is typically also on-call for it. This creates friction between adding new features and operating the service as the team needs to decide what to prioritize during each sprint.

Debugging systems failures becomes more challenging as well, as you can't just load the whole application on your local machine and step through it with a debugger. The system has more ways to fail, since there are more moving parts. This is why good logging and monitoring becomes crucial.

Eventual consistency

A side effect of splitting an application into separate services is that the data model no longer resides in a single data store. As we have learned in previous chapters, atomically updating records stored 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.

12.1.3 Practical considerations

Splitting an application into services adds a lot of complexity to the overall system. Because of that, it's generally best to start with a monolith and split

it up only when there is a good reason to do so.

Getting the boundaries right between the services is challenging — it's much easier to move them around within a monolith until you find a sweet spot. Once the monolith is well matured and growing pains start to rise, then you can start to peel off one microservice at a time from it.

You should only start with a microservice-first approach if you already have experience with it, and you either have built out a platform for it or have accounted for the time it will take you to build one.

12.2 API gateway

After you have split an application into a set of services, each with its own API, you need to rethink how clients communicate with the application. 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 very expensive on mobile devices where every network request consumes precious battery life.

Moreover, clients need to be aware of implementation details, like the DNS names of all the internal services. This makes it challenging to change the application's architecture as it could require all clients to be upgraded. To make matters worse, if clients are distributed to individual consumers (e.g., an app on the App Store), there might not be an easy way to force them all to upgrade to a new version. The bottom line is that once a public API is out there, you better be prepared to maintain it for a very long time.

As is typical in computer science, we can solve this problem by adding a layer of indirection. The internal APIs can be hidden by a public one that acts as a facade, or proxy, for the internal services (see Figure 12.3). The service that exposes the public API is called the *API gateway*, which is transparent to its clients since they have no idea they are communicating through an intermediary.

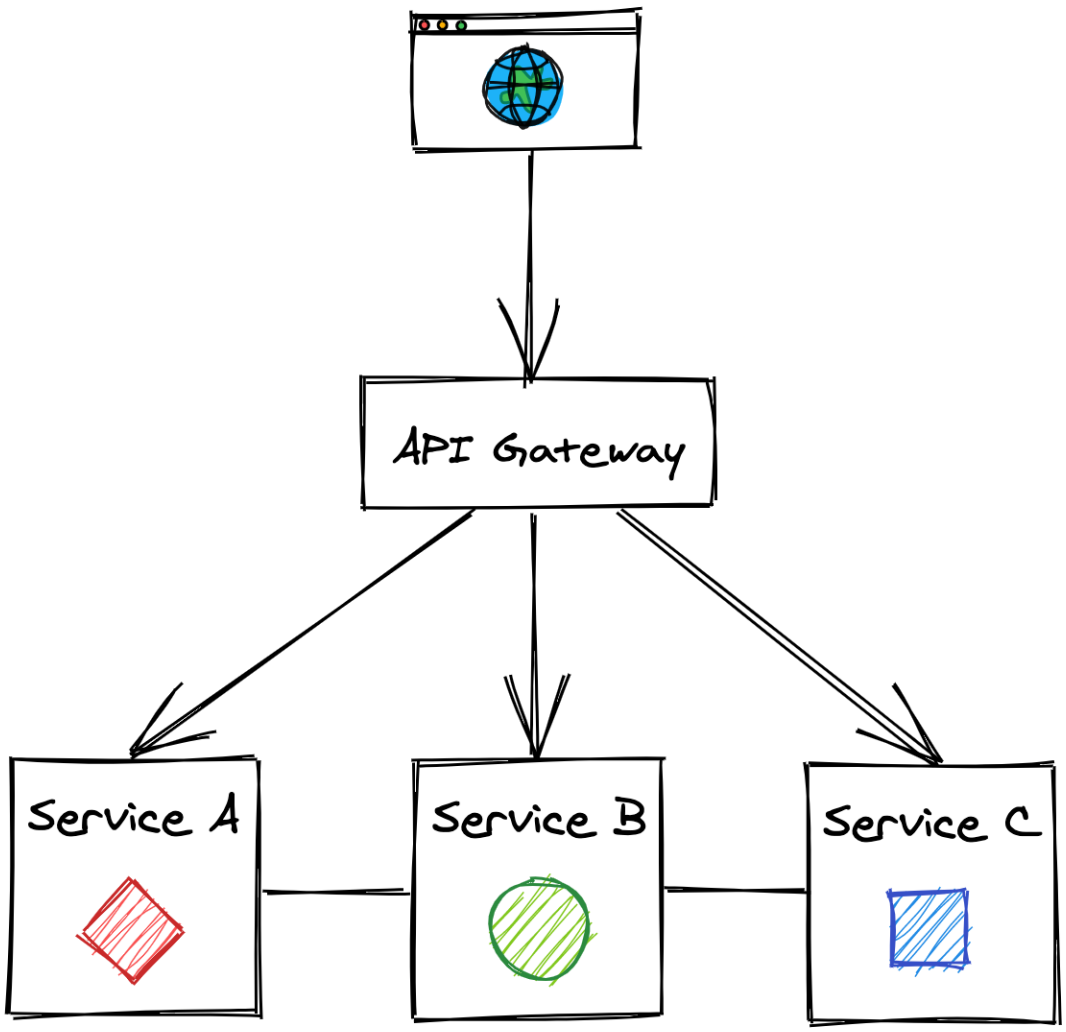


Figure 12.3: The API gateway hides the internal APIs from its clients.

The API gateway provides multiple features, like routing, composition, and translation.

12.2.1 Routing

The API gateway can route the requests it receives to the appropriate backend service. It does so with the help of a routing map, which maps the external APIs to the internal ones. For example, the map might have a 1:1 mapping between an external path and internal one. If in the future the internal path changes, the public API can continue to expose the old path to guarantee backward compatibility.

12.2.2 Composition

While data of a monolithic application typically resides in a single data store, in a distributed system, it's spread across multiple services. As such, some use cases might require stitching data back together from multiple sources. The API gateway can offer a higher-level API that queries multiple services and composes their responses within a single one that is then returned to the client. 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.

Composition can be hard to get right. The availability of the composed API decreases as the number of internal calls increases since each has a non-zero probability of failure. Additionally, the data across the services might be inconsistent as some updates might not have propagated to all services yet; in that case, the gateway will have to somehow resolve this discrepancy.

12.2.3 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.

The gateway can also expose different APIs to different types of clients. For example, a web API for a desktop application can 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 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 APIs to the internal ones. An increasingly popular approach to tailor APIs to individual use cases is to use graph-based APIs. A *graph-based API* exposes a schema composed of types, fields, and relationships across types.

The API allows a client to declare what data it needs and let the gateway figure out how to translate the request into a series of internal API calls.

This approach reduces the development time as there is no need to introduce different APIs for different use cases, and the 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. In a way, it's as if the gateway grants the clients the ability to perform restricted queries on its backend APIs. GraphQL is the most popular technology in the space at the time of writing.

12.2.4 Cross-cutting concerns

As the API gateway is a proxy, or middleman, for the services behind it, it can also implement cross-cutting functionality that otherwise would have to be re-implemented in each service. For example, the API gateway could cache frequently accessed resources to improve the API's performance while reducing the bandwidth requirements on the services or rate-limit requests to protect the services from being overwhelmed.

Among the most critical cross-cutting aspects of securing a service, authentication and authorization are top-of-mind. *Authentication* is the process of validating that a so-called principal — a human or an application — issuing a request from a client is who it says it is. *Authorization* instead 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 a principal one or more roles that grant specific permissions. Alternatively, an access control list can be used to grant specific principals access to specific resources.

A monolithic application can implement authentication and authorization with session tokens. A client sends its credentials to the application API's login endpoint, which validates the credentials. If that's successful, the endpoint returns a *session token*[1] to the client, typically through an HTTP cookie. The client then includes the token in all future requests.

The application uses the session token to retrieve a session object from an in-memory cache or an external data store. The object contains the

principal's ID and the roles granted to it, which are used by the application's API handlers to decide whether to allow the principal to perform an operation 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 be responsible for authenticating 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 their domain logic.

When the API gateway has authenticated a request, it creates a *security token*. The gateway passes this token to the internal services responsible for handling the request, which in turn will pass it downstream to their dependencies (see Figure 12.4).

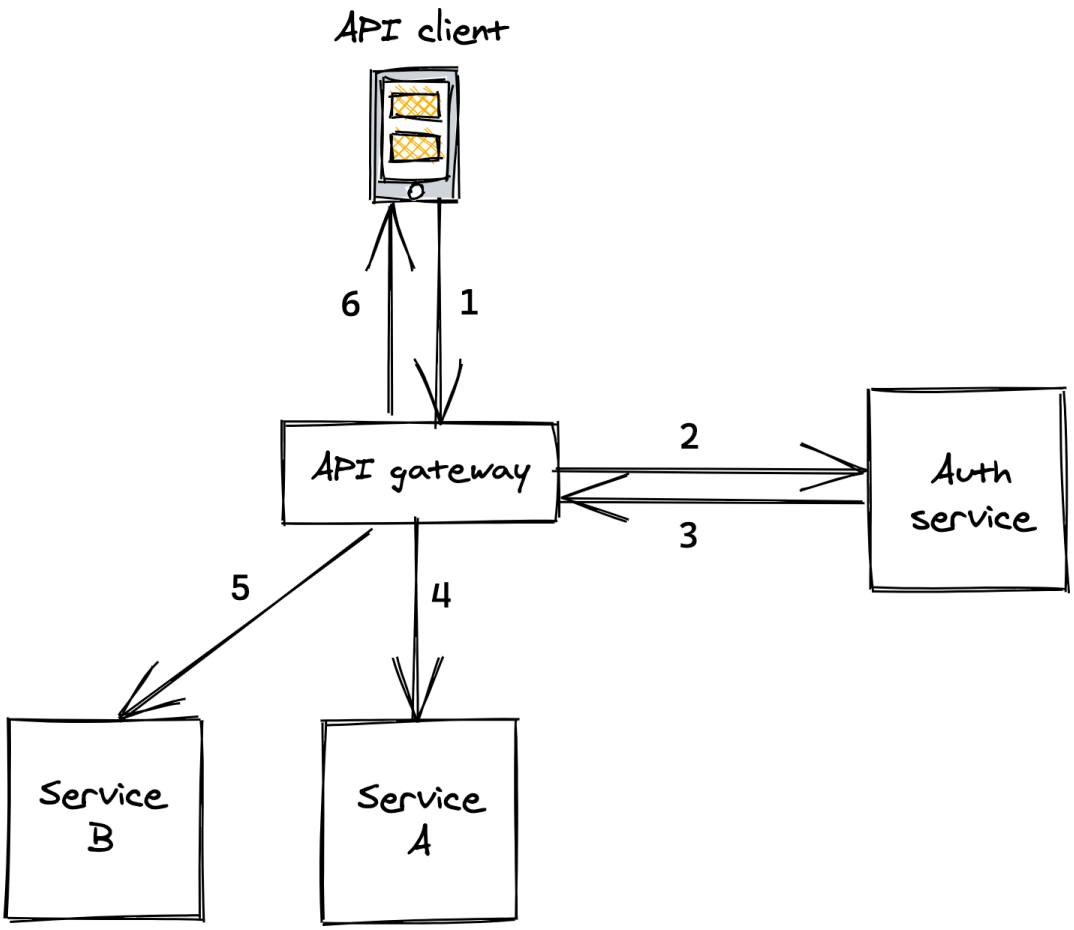


Figure 12.4: . API client sends a request with credentials to API gateway . API gateway tries to authenticate credentials with auth service . Auth service validates credentials and replies with a security token . API gateway sends a request to service A including the security token . API gateway sends a request to service B including the security token . API gateway composes the responses from A and B and replies to the client

When an internal service receives a request with a security token attached to it, it needs to have a way to validate it and obtain the principal's identity and its roles. The validation differs depending on the type of token used, which can be either opaque and not contain any information (e.g., an UUID), or be transparent and embed the principal's information within the token itself.

The downside of opaque tokens is that they require services to call an external auth service to validate a token and retrieve the principal's information. Transparent tokens eliminate that call at the expense of making it harder to revoke issued tokens that have fallen into the wrong hands.

The most popular standard for transparent tokens is the *JSON Web Token* (JWT). A JWT is a JSON payload that contains an expiration date, the principal's identity, roles, and other metadata. The payload is signed with a certificate trusted by internal services. Hence, no external calls are needed to validate the token.

OpenID Connect and OAuth 2 are security protocols that you can use to implement token-based authentication and authorization. We have barely scratched the surface on the topic, and there are entire books written on the subject you can read to learn more about it.

Another widespread mechanism to authenticate applications is the use API keys. An API key is a custom key that allows the API gateway to identify which application is making a request and limit what they can do. This approach is popular for public APIs, like the one offered by Github or Twitter.

12.2.5 Caveats

One of the drawbacks of using an API gateway is that it can become a development bottleneck. As it's coupled with the services it's hiding, every new service that is created needs to be wired up to it. Additionally, whenever the API of service changes, the gateway needs to be modified as well.

The other downside is that the API gateway is one more service that needs to be developed, maintained, and operated. Also, it needs to be able to scale to whatever the request rate is for all the services behind it. That said, if an application has dozens of services and APIs, the upside is greater than the downside 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 proxy framework as a starting point, like NGINX. Or

better yet, you can use an off-the-shelf solution, like Azure API Management.

12.3 CQRS

The API's gateway ability to compose internal APIs is quite limited, and querying data distributed across services can be very inefficient if the composition requires large in-memory joins.

Accessing data can also be inefficient for reasons that have nothing to do with using a microservice architecture:

- The data store used might not be well suited for specific types of queries. For example, a vanilla relational data store isn't optimized for geospatial queries.
- The data store might not scale to handle the number of reads, which could be several orders of magnitude higher than the number of writes.

In these cases, decoupling the read path from the write path can yield substantial benefits. This approach is also referred to as the *Command Query Responsibility Segregation* (CQRS) pattern.

The two paths can use different data models and data stores that fit their specific use cases (see Figure 12.5). For example, the read path could use a specialized data store tailored to a particular query pattern required by the application, like geospatial or graph-based.

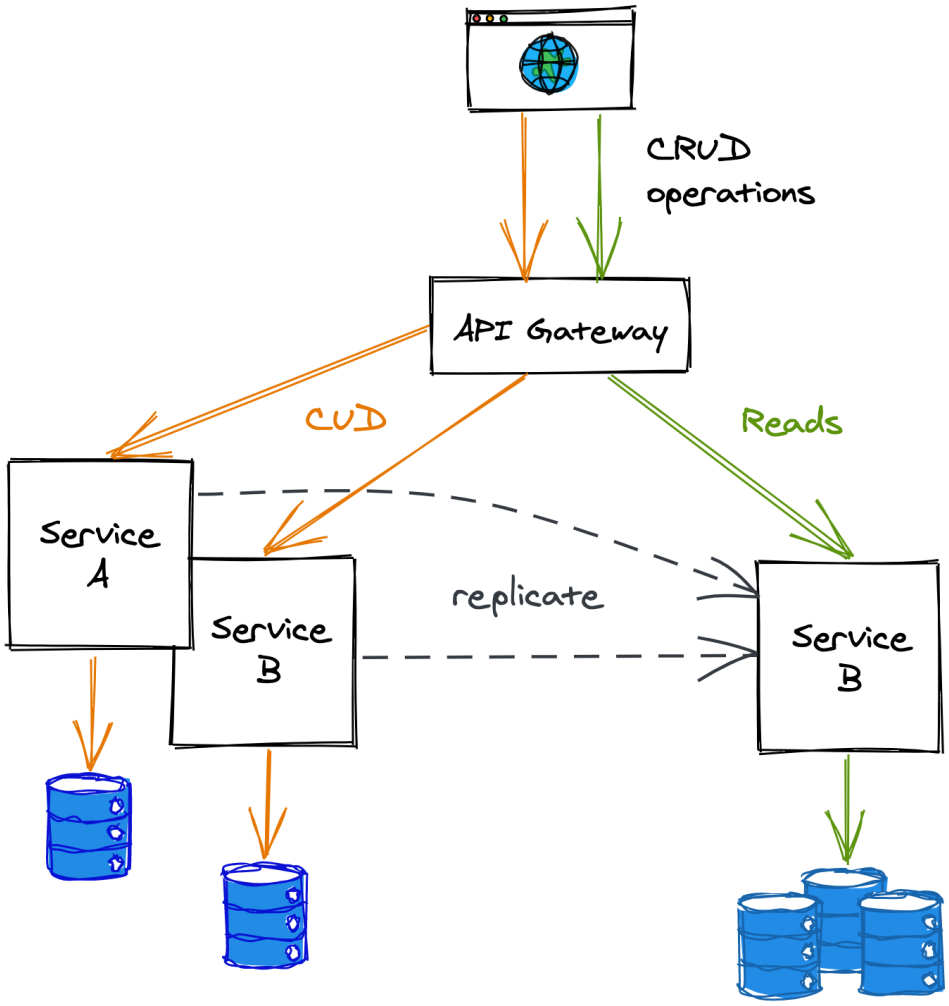


Figure 12.5: In this example, the read and write paths are separated out into different services.

To keep the read and write data models synchronized, the write path pushes updates to the read path whenever the data changes. External clients could still use the write path for simple queries, but complex queries are routed to the read path.

This separation adds more complexity to the system. For example, when the data model changes, both paths might need to be updated. Similarly, operational costs increase as there are more moving parts to maintain and operate. Also, there is an inherent replication lag between the time a change has been applied on the write path and the read path has received and applied it, which makes the system sequentially consistent.

12.4 Messaging

When an application is decomposed into services, the number of network calls increases, and with it, the probability that a request's destination is momentarily unavailable. So far, we have mostly assumed services communicate using a direct request-response communication style, which requires the destination to be available and respond promptly. Messaging — a form of indirect communication — doesn't have this requirement, though.

Messaging was first introduced when we discussed the implementation of asynchronous transactions in section 11.4.1. It is 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.

By decoupling the producer from the consumer, the former gains the ability to communicate with the latter even if it's temporarily unavailable. Messaging provides several other benefits:

- It allows a client to execute an operation on a service asynchronously. This is particularly convenient for operations that can take a long time to execute. For example, suppose a video needs to be converted to multiple formats optimized for different devices. In that case, the client could write a message to a channel to trigger the conversion and be done with it.
- It can load balance messages across a pool of consumers, which can dynamically be added or removed to match the current load.
- It helps to smooth out load spikes allowing consumers to read messages at their own pace without getting overloaded.

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. Additionally, the system's complexity increases as there is one more service, the message broker, that needs to be maintained and operated — as always, it's all about tradeoffs.

Any number of producers can write messages to a channel, and similarly, multiple consumers can read from it. Depending on how the channel delivers messages to consumers, it can be classified as either point-to-point or publish-subscribe. In a *point-to-point* channel, a specific message is delivered to exactly one consumer. Instead, in a *publish-subscribe* channel, a copy of the same message is delivered to all consumers.

A message channel can be used for a variety of different communication styles.

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 12.6).

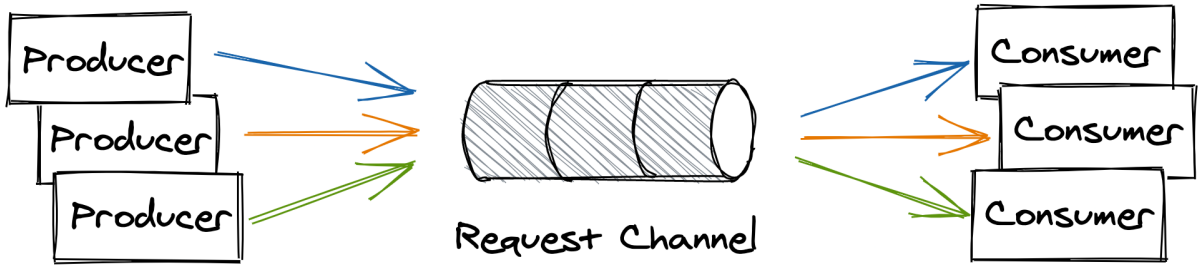


Figure 12.6: One-way messaging style

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 own dedicated response channel (see Figure 12.7).

When a producer writes a message to the request channel, it decorates it with a request id and a reference to its response channel. 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.

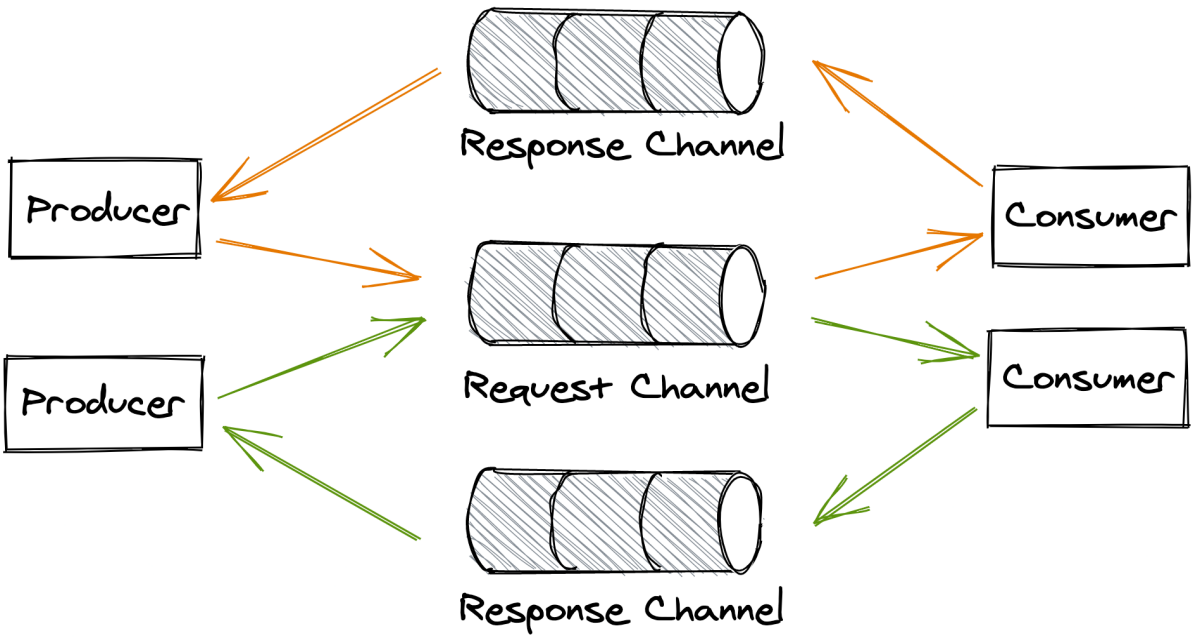


Figure 12.7: Request-response messaging style

Broadcast messaging

In this messaging style, a producer writes a message to a publish-subscribe channel to broadcast it to all consumers (see Figure 12.8). This mechanism is generally used to notify a group of processes that a specific event has occurred. We have already encountered this pattern when discussing log-based transactions in section 11.4.1.

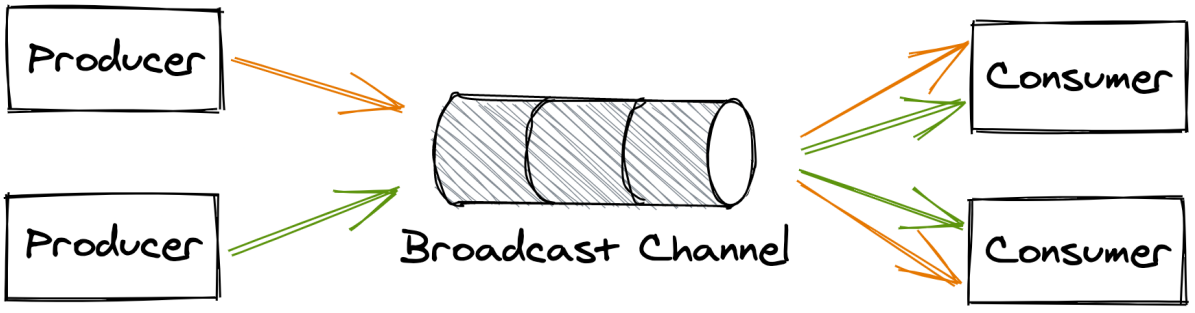


Figure 12.8: Broadcast messaging style

12.4.1 Guarantees

A message channel is implemented by a messaging service, like AWS SQS or Kafka. The messaging service, or broker, acts as a buffer for messages. It decouples producers from consumers so that they don't need to know the consumers' addresses, how many of them there are, or whether they are available.

Different message brokers implement the channel abstraction differently depending on the tradeoffs and the guarantees they offer. 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, don't offer any strong ordering guarantees. Why is that?

Because a message broker needs to scale out just like 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, each small enough to be handled entirely by a single process. The idea is that if there is a single broker process responsible for the messages of a sub-channel, then it should be trivial to guarantee their order.

In this case, when messages are sent to the channel, they are partitioned into sub-channels based on a partition key. To guarantee that the message order is preserved end-to-end, only a single consumer process can be allowed to read from a sub-channel[2].

Because the channel is partitioned, it suffers from several drawbacks. For example, a specific partition can become much hotter than the others, and the single consumer reading from it might not be able to keep up with the load. In that case, the channel needs to be repartitioned, which can temporarily degrade the broker since messages need to be reshuffled across all partitions. Later in the chapter, we will learn more about the pros and cons of partitioning.

Now you see why not having to guarantee 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;
- support for competing consumers;
- 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:

- Channels are point-to-point and support an arbitrary number of producers and consumers.
- Messages are delivered to consumers at-least-once.
- While a consumer is processing a message, the message remains persisted in the channel, but other consumers can't read it for the duration of a visibility timeout. The *visibility timeout* guarantees that if the consumer crashes while processing the message, the message will become visible to other consumers again when the timeout triggers. When the consumer is done processing the message, it deletes it from the channel preventing it from being received by any other consumer in the future.

The above guarantees are very similar to what cloud services such as Amazon's SQS and Azure Storage Queues offer.

12.4.2 Exactly-once processing

As mentioned, a consumer has to delete a message from the channel once it's done processing it so that it won't be read by another consumer.

If the consumer deletes the message before processing it, there is a risk it could crash after deleting the message and before processing it, causing the message to be lost for good. On the other hand, if the consumer deletes the message only after processing it, there is a risk that the consumer might crash 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 as *exactly-once message delivery*. The best a consumer can do is to simulate *exactly-once message processing* by requiring messages to be idempotent.

12.4.3 Failures

When a consumer fails to process a message, the visibility timeout triggers, and the message is eventually delivered to another consumer. 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, it can be implemented by the consumers.

Once you have a way to count the number of times a message has been retried, you 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 acts as a buffer for 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 consumers' processing 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.

12.4.4 Backlogs

One of the main advantages of using a messaging broker is that it makes the system more robust to outages. Producers can continue to write messages to a channel even if one or more consumers are not available or are degraded. As long as the rate of arrival of messages is lower or equal to the rate they

are being deleted from the channel, everything is great. When that is no longer true, and consumers can't keep up with producers, a backlog starts to build up.

A messaging channel introduces a bi-modal 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 producers came online, and/or their throughput increased, and the consumers can't match their rate;
- the consumers have become slower to process individual messages, which in turn decreased their deletion rate;
- the consumers fail to process a fraction of the messages, which are picked up again by other consumers until they eventually end up in the dead letter channel. This can cause a negative feedback loop that delays healthy messages and wastes the consumers' processing time.

To detect backlogs, you should 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), the measure still provides a good indication of the backlog.

12.4.5 Fault isolation

A specific producer that emits "poisonous" messages that repeatedly fail to be processed can degrade the whole system and potentially cause backlogs, since messages are processed multiple times before they end up in the dead-letter channel. Therefore, it's important to find ways to deal with problematic producers before they start to affect the rest of the system[3].

If messages belong to different users[4] and are decorated with some kind of identifier, consumers can decide to treat "noisy" users differently. For example, suppose messages from a specific user fail consistently. In that case, the consumers could decide to write these messages to an alternate low-priority channel and remove them from the main channel without processing them. The consumers can continue to read from the slow channel, but do so less frequently. This ensures that one single bad user can't affect others.

12.4.6 Reference plus blob

Transmitting a large binary object (blob) like images, audio files, or video can be challenging or simply impossible, depending on the medium. For example, message brokers limit the maximum size of messages that can be written to a channel; Azure Storage queues limit messages to 64 KB, AWS Kinesis to 1 MB, etc. So how do you transfer large blobs of hundreds of MBs with these strict limits?

You can upload a blob to an object storage service, like AWS S3 or Azure Blob Storage, and then send the URL of the blob via message (this pattern is sometimes referred to as queue plus blob). The downside is that now you have to deal with two services, the message broker and the object store, rather than just the message broker, which increases the system's complexity.

A similar approach can be used to store large blobs in databases — rather than storing a blob in a database directly, you only store some metadata containing an external reference to the actual blob. The advantage of this solution is that it minimizes data being transferred back and forth to and from the data store, improving its performance while reducing the required bandwidth. Also, the cost per byte of an object store designed to persist large objects that infrequently change, if at all, is lower than the one of a generic data store.

Of course, the downside is that you lose the ability to transactionally update the blob with its metadata and potentially other records in the data store. For example, suppose a transaction inserts a new record in the data store

containing an image. In this case, the image won't be visible until the transaction completes; that won't be the case if the image is stored in an external store, though. Similarly, if the record is later deleted, the image is automatically deleted as well; but if the image lives outside the store, it's your responsibility to delete it.

Whether storing blobs outside of your data store is acceptable or not depends on your specific use cases.

---

1. e.g., a cryptographically-strong random number↩

2. This is also referred to as the *competing consumer pattern*, which is implemented using leader election↩

3. These producers are also referred to as noisy neighbors↩

4. A user can be a human or an application.↩

13 Partitioning

Now it's time to change gears and dive into another tool you have at your disposal to scale out application — partitioning or sharding.

When a dataset no longer fits on a single node, it needs to be partitioned across multiple nodes. Partitioning is a general technique that can be used in a variety of circumstances, like sharding TCP connections across backends in a load balancer. To ground the discussion in this chapter, we will anchor it to the implementation of a sharded key-value store.

13.1 Sharding strategies

When a client sends a request to a partitioned data store to read or write a key, the request needs to be routed to the node responsible for the partition the key belongs to. One way to do that is to use a gateway service that can route the request to the right place knowing how keys are mapped to partitions and partitions to nodes.

The mapping between keys and partitions, and other metadata, is typically maintained in a strongly-consistent configuration store, like etcd or Zookeeper. But how are keys mapped to partitions in the first place? At a high level, there are two ways to implement the mapping using either range partitioning or hash partitioning.

13.1.1 Range partitioning

With range partitioning, the data is split into partitions by key range in lexicographical order, and each partition holds a continuous range of keys, as shown in Figure 13.1. The data can be stored in sorted order on disk within each partition, making range scans fast.

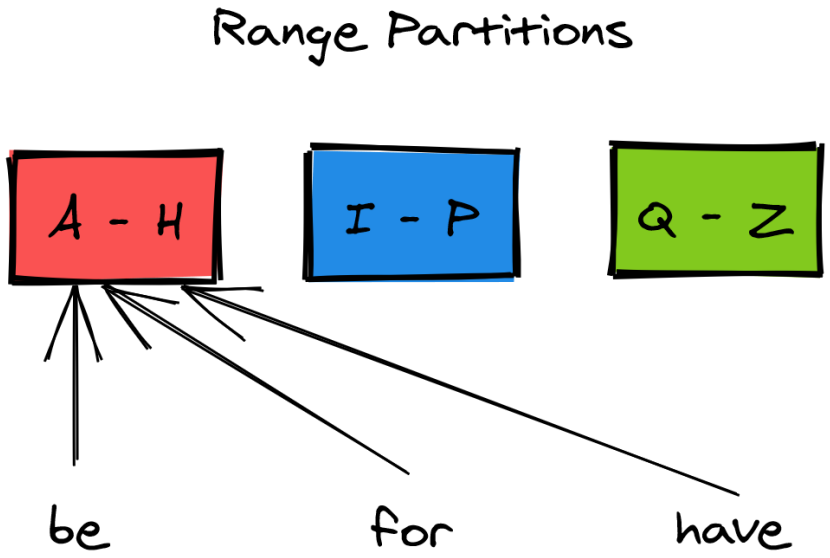


Figure 13.1: A range partitioned dataset

Splitting the key-range evenly doesn't make much sense though if the distribution of keys is not uniform, like in the English dictionary. Doing so creates unbalanced partitions that contain significantly more entries than others.

Another issue with range partitioning is that some access patterns can lead to hotspots. For example, if a dataset is range partitioned by date, all writes for the current day end up in the same partition, which degrades the data store's performance.

13.1.2 Hash partitioning

The idea behind hash partitioning is to use a hash function to assign keys to partitions, which shuffles — or uniformly distributes — keys across partitions, as shown in Figure 13.2. Another way to think about it is that the hash function maps a potentially non-uniformly distributed key space to a uniformly distributed hash space.

For example, a simple version of hash partitioning can be implemented with modular hashing, i.e., hash(key) mod N.

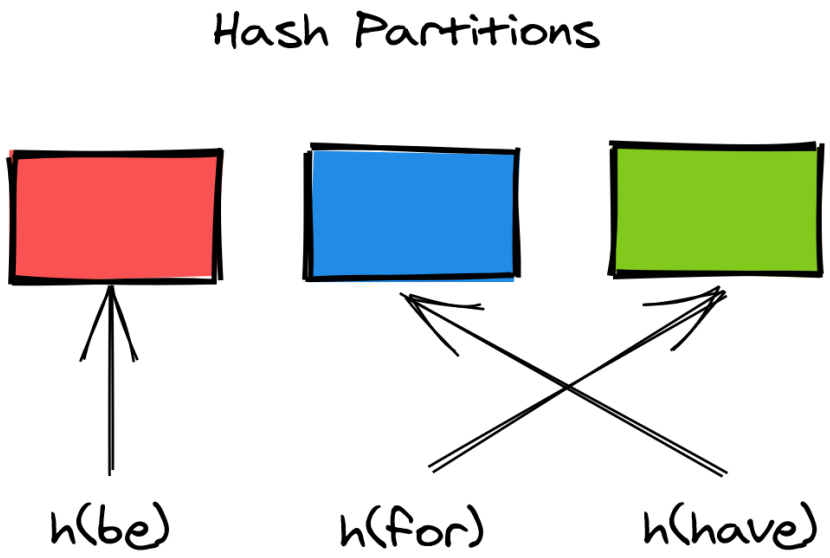


Figure 13.2: A hash partitioned dataset

Although this approach ensures that the partitions contain more or less the same number of entries, it doesn't eliminate hotspots if the *access pattern* is not uniform. If there is a single key that is accessed significantly more often than others, then all bets are off. In this case, the partition that contains the hot key needs to be split further down. Alternatively, the key needs to be split into multiple sub-keys, for example, by adding an offset at the end of it.

Using modular hashing can become problematic when a new partition is added, as all keys have to be reshuffled across partitions. Shuffling data is extremely expensive as it consumes network bandwidth and other resources from nodes hosting partitions. Ideally, if a partition is added, only $\frac{K}{N}$ keys should be shuffled around, where K is the number of keys and N the number of partitions. A hashing strategy that guarantees this property is called stable hashing.

Ring hashing is an example of stable hashing. With ring hashing, a function maps a key to a point on a circle. The circle is then split into partitions that can be evenly or pseudo-randomly spaced, depending on the specific

algorithm. When a new partition is added, it can be shown that most keys don't need to be shuffled around.

For example, with *consistent hashing*, both the partition identifiers and keys are randomly distributed on a circle, and each key is assigned to the next partition that appears on the circle in clockwise order (see Figure 13.3).

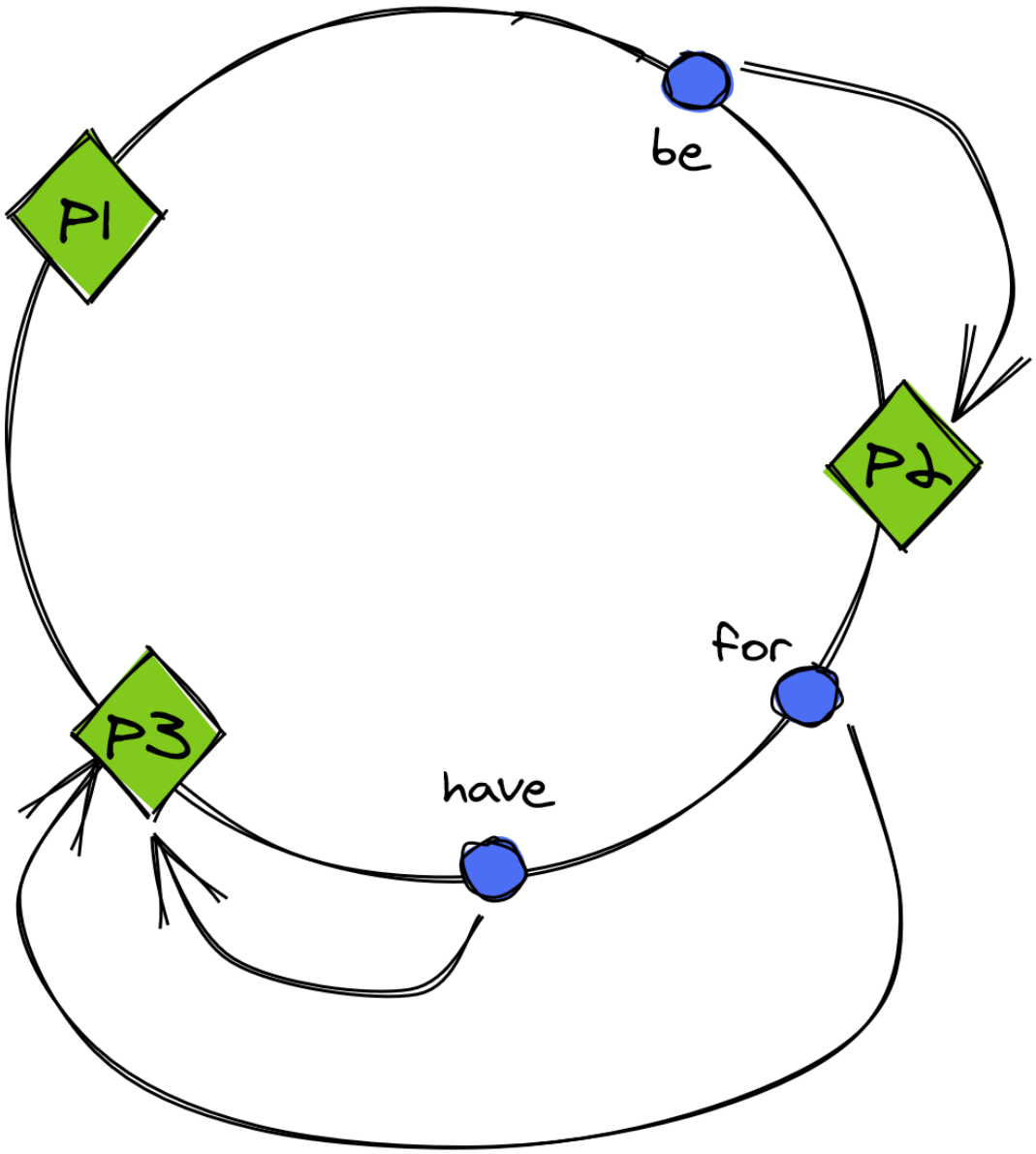


Figure 13.3: With consistent hashing, partition identifiers and keys are randomly distributed on a circle, and each key is assigned to the next

partition that appears on the circle in clockwise order.

Now, when a new partition is added, only the keys mapped to it need to be reassigned, as shown in Figure 13.4.

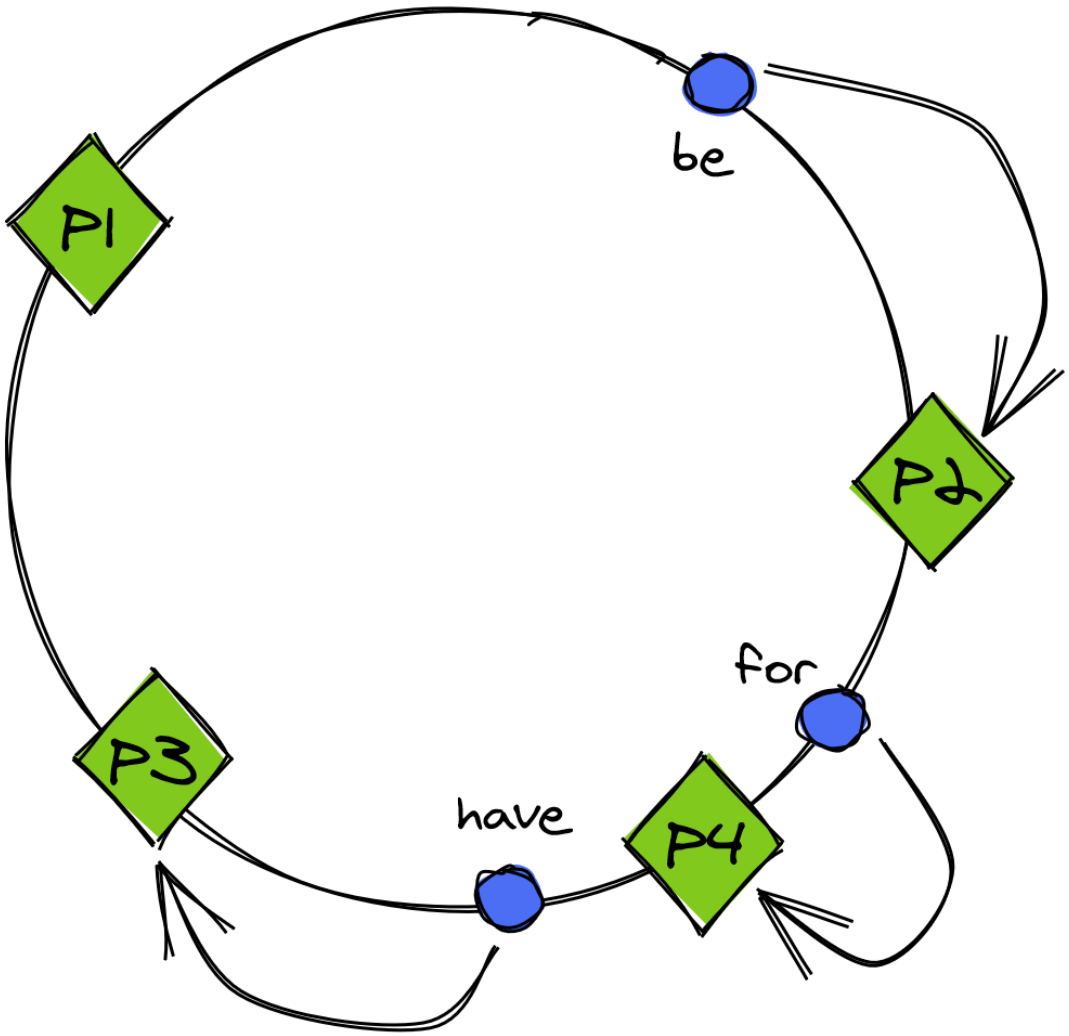


Figure 13.4: After partition P4 is added, key ‘for’ is reassigned to P4, but the other keys are not reassigned.

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

13.2 Rebalancing

When the number of requests to the data store becomes too large, or the dataset's size becomes too large, the number of nodes serving partitions needs to be increased. Similarly, if the dataset's size keeps shrinking, the number of nodes can be decreased to reduce costs. The process of adding and removing nodes to balance the system's load is called rebalancing.

Rebalancing needs to be implemented in such a way to minimize disruption to the data store, which needs to continue to serve requests. Hence, the amount of data transferred during the rebalancing act needs to be minimized.

13.2.1 Static partitioning

Here, the idea is to create way more partitions than necessary when the data store is first initialized and assign multiple partitions per node. 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 set when the data store is first initialized and can't be easily changed after that. Getting the number of partitions wrong can be problematic — too many partitions add overhead and decrease the data store's performance, while too few partitions limit the data store's scalability.

13.2.2 Dynamic partitioning

An alternative to creating partitions upfront is to create them on-demand. One way to implement dynamic partitioning is to start with a single partition. When it grows above a certain size or becomes too hot, it's split into two sub-partitions, each containing approximately half of the data. Then, one sub-partition is transferred to a new node. Similarly, if two adjacent partitions become small enough, they can be merged into a single one.

13.2.3 Practical considerations

Introducing partitions in the system adds a fair amount of complexity, even if it appears deceptively simple. Partition imbalance can easily become a headache as a single hot partition can bottleneck the system and limit its ability to scale. And as each partition is independent of the others, transactions are required to update multiple partitions atomically.

We have merely scratched the surface on the topic; if you are interested to learn more about it, I recommend reading Designing Data-Intensive Applications by Martin Kleppmann.

14 Duplication

Now it's time to change gears and dive into another tool you have at your disposal to design horizontally scalable applications — duplication.

14.1 Network load balancing

Arguably the easiest way to add more capacity to a service is to create more instances of it and have some way of routing, or balancing, requests to them. The thinking is that if one instance has a certain capacity, then 2 instances should have a capacity that is twice that.

Creating more service instances can be a fast and cheap way to scale out a stateless service, as long as you have taken into account the impact on its dependencies. For example, if every service instance needs to access a shared data store, eventually, the data store will become a bottleneck, and adding more service instances to the system will only strain it further.

The routing, or balancing, of requests across a pool of servers is implemented by a network load balancer. A *load balancer* (LB) has one or more physical *network interface cards* (NIC) mapped to one or more *virtual IP* (VIP) addresses. A VIP, in turn, is associated with a pool of servers. The LB acts as a middle-man between clients and servers — the clients only see the VIP exposed by the LB and have no visibility of the individual servers associated with it.

Distributing requests across 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 LB can be increased or reduced transparently. And since multiple redundant servers can interchangeably be used to handle requests, a LB can detect faulty ones and take them out of the pool, increasing the service's availability.

At a high level, a LB supports several core features beyond load balancing, like service discovery and health-checks.

Load Balancing

The algorithms used for routing requests can vary from simple round-robin to more complex ones that take into account the servers' load and health. There are several ways for a LB to infer the load of the servers. For example, the LB could periodically hit a dedicated *load endpoint* of each server that returns a measure of how busy the server is (e.g., CPU usage). Hitting the servers constantly can be very costly though, so typically a LB caches these measures for some time.

Using cached, and hence delayed, metrics to distribute requests to servers can create a herding effect. Suppose the load metrics are refreshed periodically, and a server that just joined the pool reported a load of 0 — guess what happens next? The LB is going to hammer that server until the next time its load is sampled. When that happens, the server is marked as busy, and the LB stops sending more requests to it, assuming it hasn't become unavailable first due to the volume of requests sent its way. This creates a ping-pong effect where the server alternates between being very busy and not busy at all.

Because of this herding effect, it turns out that randomly distributing requests to servers without accounting for their load actually achieves a better load distribution. Does that mean that load balancing using delayed load metrics is not possible?

Actually, 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 as it combines delayed load information with the protection against herding that randomness provides.

Service Discovery

Service discovery is the mechanism used by the LB to discover the available servers in the pool it can route requests to. There are various ways to implement it. For example, a simple approach is to use a static configuration file that lists the IP addresses of all the servers. However, this is quite painful to manage and keep up-to-date. A more flexible solution can

be implemented with DNS. Finally, using a data store provides the maximum flexibility at the cost of increasing the system's complexity.

One of the benefits of using a dynamic service discovery mechanism is that servers can be added and removed from the LB's pool at any time. This is a crucial functionality that cloud providers leverage to implement autoscaling, i.e., the ability to automatically spin up and tear down servers based on their load.

Health checks

Health checks are used by the LB 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 LB 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 LB can decide to take that server out from the pool.

Instead, an *active health check* requires support from the downstream servers, which need to expose a *health endpoint* signaling the server's health state. Later in the book, we will describe in greater detail how to implement such a health endpoint.

14.1.1 DNS load balancing

Now that we know what a load balancer's job is, let's take a closer look at how it can be implemented. While you probably won't have to build your own LB given the plethora of off-the-shelf solutions available, a basic knowledge of how load balancing works is crucial. LB failures are very visible to your services' clients since they tend to manifest themselves as timeouts and connection resets. Because the LB sits between your service and its clients, it also contributes to the end-to-end latency of request-response transactions.

The most basic form of load balancing can be implemented with DNS. Suppose you have a couple of servers that you would like to load balance

requests over. If these servers have publicly-reachable IP addresses, you can add those to the service's DNS record and have the clients pick one when resolving the DNS address, as shown in Figure 14.1.

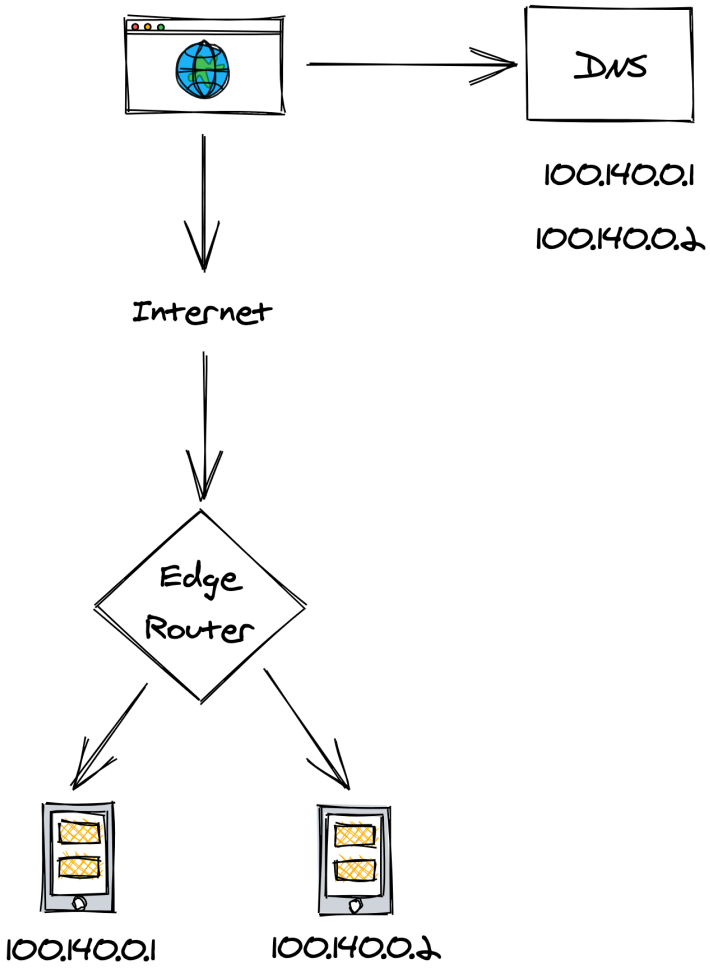


Figure 14.1: DNS load balancing

Although this works, it doesn't deal well with failures. If one of the two servers goes down, the DNS server will happily continue serving its IP address unaware of the failure. You can manually reconfigure the DNS record to take out the problematic IP, but as we have learned in chapter 4, changes are not applied immediately due to the nature of DNS caching.

14.1.2 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[1], through which all traffic between clients and servers needs to go through.

When a client creates a new TCP connection with a LB's VIP, the LB picks a server from the pool and henceforth shuffles the packets back and forth for that connection between the client and the server. How does the LB assign connections to the servers, though?

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. To minimize the disruption caused by a server being added or removed from the pool, consistent hashing is preferred over modular hashing.

To forward packets downstream, the LB translates each packet's source address to the LB address and its destination address to the server's address. Similarly, when the LB receives a packet from the server, it translates its source address to the LB address and its destination address to the client's address (see Figure 14.2).

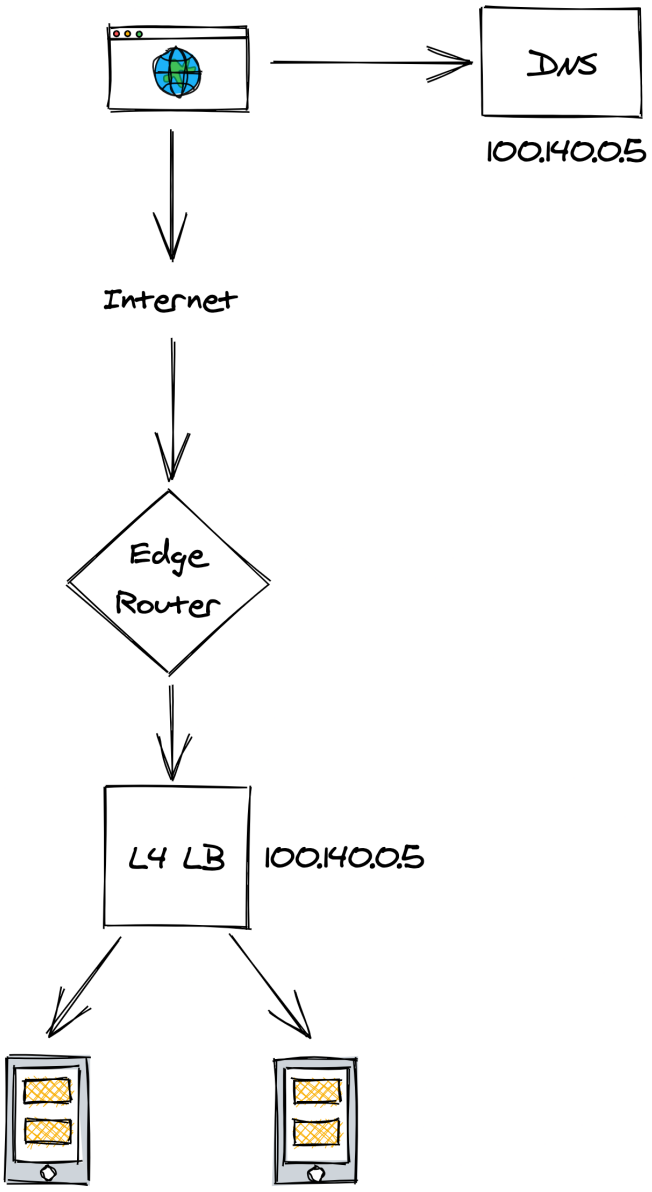


Figure 14.2: Transport layer load balancing

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 LB and respond directly to the clients using a mechanism called direct server return, but this is beyond the scope of this section.

Because the LB is communicating directly with the servers, it can detect unavailable ones (e.g., with a passive health check) and automatically take them out of the pool improving the reliability of the backend service.

Although load balancing connections at the TCP level is very fast, the drawback is that the LB is just shuffling bytes around without knowing what they actually mean. Therefore, L4 LBs generally don't support features that require higher-level network protocols, like terminating TLS connections or balancing HTTP sessions based on cookies. A load balancer that operates at a higher level of the network stack is required to support these advanced use cases.

14.1.3 Application layer load balancing

An application layer load balancer[2] is an HTTP reverse proxy that farms out requests over a pool of servers. The LB 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 LB and another between the L7 LB and the server. Because a L7 LB 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 several orders of magnitude more expensive to handle than others.

The LB can do smart things with application traffic, like rate-limiting 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 LB could use a specific cookie to identify which logical session a specific request belongs to. Just like with a L4 LB, the session identifier can be mapped to a server using consistent hashing. The caveat is that sticky sessions can create hotspots as some sessions are more expensive to handle than others.

If it sounds like a L7 LB has some overlapping functionality with an API gateway, it's because they both are HTTP proxies, and therefore their responsibilities can be blurred.

A L7 LB is typically used as the backend of a L4 LB to load balance requests sent by external clients from the internet (see Figure 14.3). Although L7 LBs offer more functionality than L4 LBs, they have a lower throughput in comparison, which makes L4 LBs better suited to protect against certain DDoS attacks, like SYN floods.

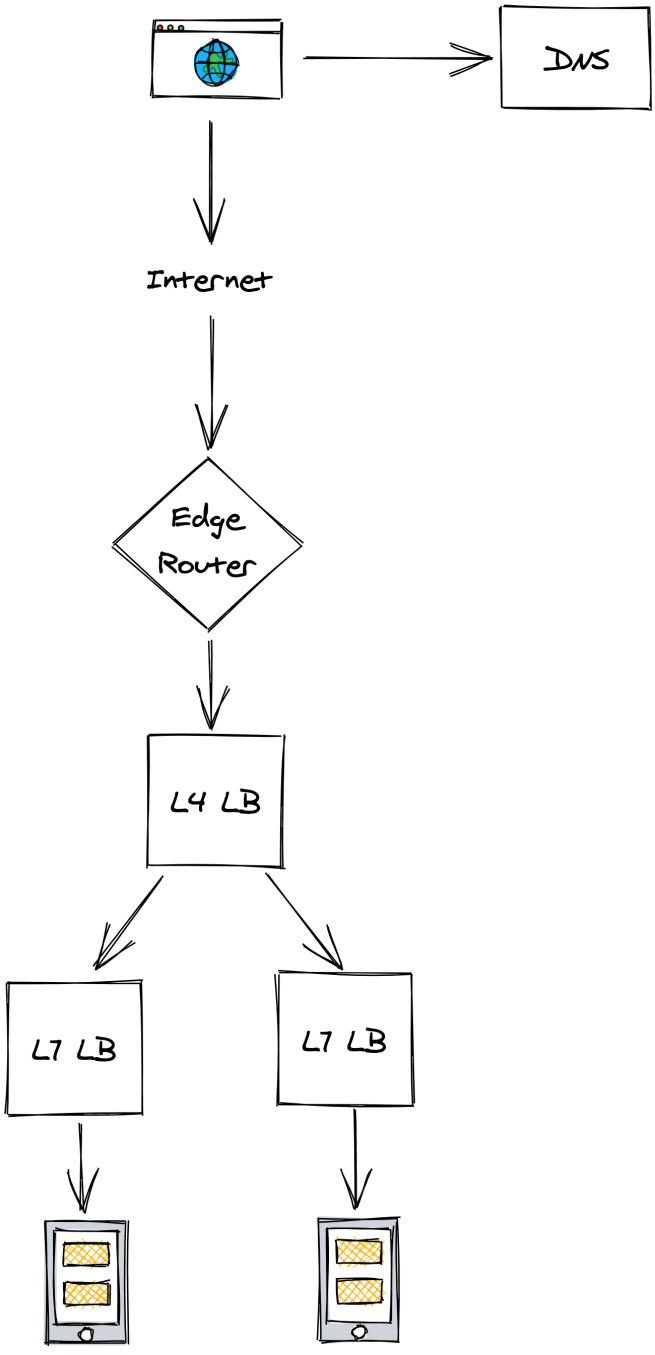


Figure 14.3: A L7 LB is typically used as the backend of a L4 one to load balance requests sent by external clients from the internet.

A drawback of using a dedicated load-balancing service is that all the traffic needs to go through it and if the LB goes down, the service behind it is no longer reachable. Additionally, it's one more service that needs to be operated and scaled out.

When the clients are internal to an organization, the L7 LB functionality can alternatively be bolted onto the clients directly using the *sidecar pattern*. In this pattern, all network traffic from a client goes through a process co-located on the same machine. This process implements load balancing, rate-limiting, authentication, monitoring, and other goodies.

The sidecar processes form the data plane of a *service mesh*, which is configured by a corresponding control plane. This approach has been gaining popularity with the rise of microservices in organizations that have hundreds of services communicating with each other. Popular sidecar proxy load balancers as of this writing are NGINX, HAProxy, and Envoy. The advantage of using this approach is that it distributes the load-balancing functionality to the clients, removing the need for a dedicated service that needs to be scaled out and maintained. The con is a significant increase in the system's complexity.

14.1.4 Geo load balancing

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.

To mitigate these performance issues, you can distribute the traffic to different data centers located in different regions. But how do you ensure

that the clients communicate with the geographically closest L4 load balancer?

This is where DNS geo load balancing comes in — it's an extension to DNS that considers the location of the client inferred from its IP, and returns a list of the geographically closest L4 LB VIPs (see Figure 14.4). The LB also needs to take into account the capacity of each data center and its health status.

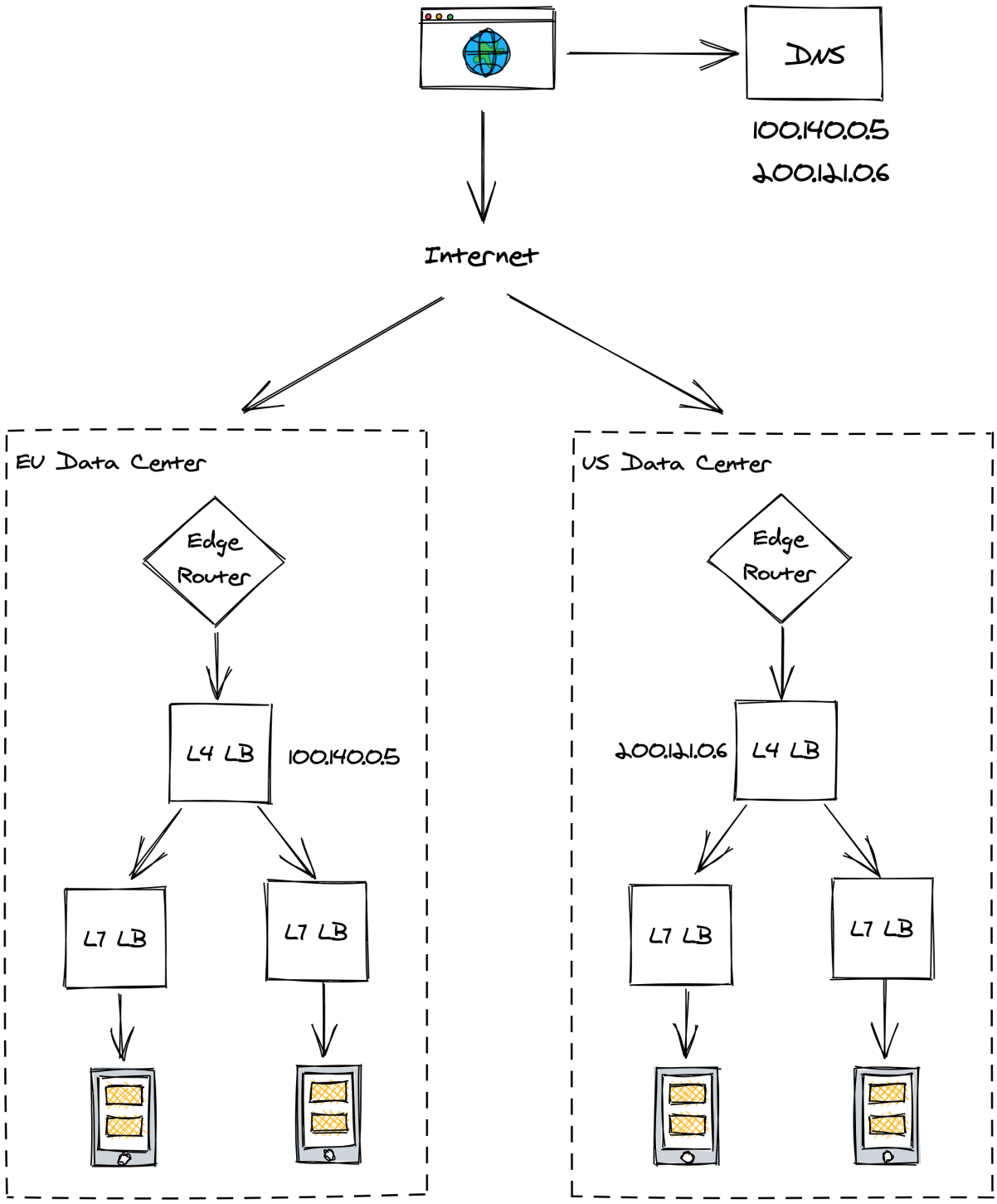


Figure 14.4: Geo load balancing infers the location of the client from its IP

14.2 Replication

If the servers behind a load balancer are stateless, scaling out is as simple as adding more servers. But when there is state involved, some form of coordination is required.

Replication is the process of storing a copy of the same data in multiple nodes. If the data is static, replication is easy: just copy the data to multiple nodes, add a load balancer in front of it, and you are done. The challenge is dealing with dynamically changing data, which requires coordination to keep it in sync.

Replication and sharding are techniques that are often combined, but are orthogonal to each other. For example, a distributed data store can divide its data into N partitions and distribute them over K nodes. Then, a state-machine replication algorithm like Raft can be used to replicate each partition R times (see Figure 14.5).

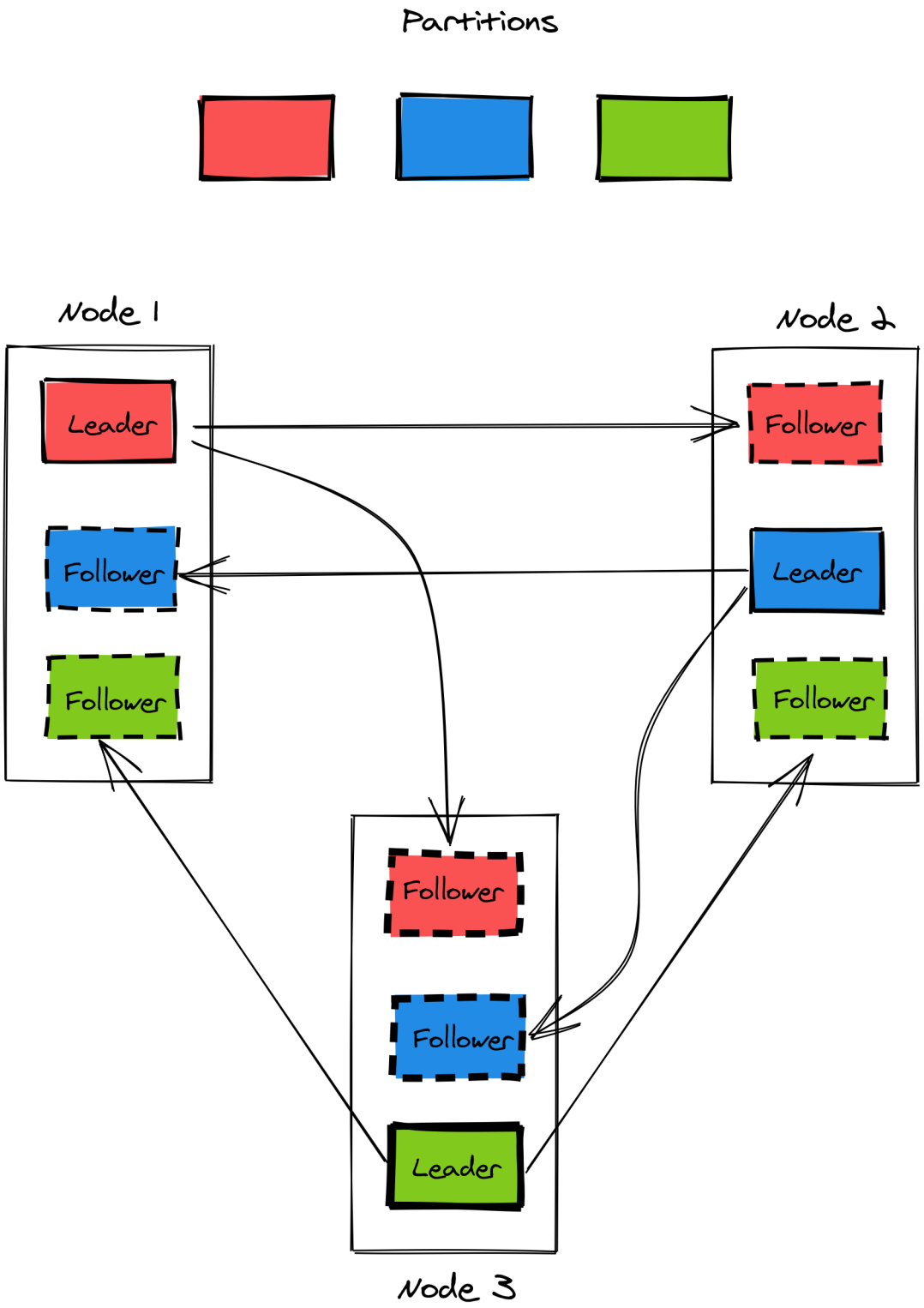


Figure 14.5: A replicated and partitioned data store. A node can be the replication leader for a partition while being a follower for another one.

We have already discussed one way of replicating data in chapter 10. This section will take a broader, but less detailed, look at replication and explore different approaches with varying trade-offs. To keep things simple, we will assume that the dataset is small enough to fit on a single node, and therefore no partitioning is needed.

14.2.1 Single leader replication

The most common approach to replicate data is the single leader, multiple followers/replicas approach (see Figure 14.6). In this approach, the clients send writes exclusively to the leader, which updates its local state and replicates the change to the followers. We have seen an implementation of this when we discussed the Raft replication algorithm.

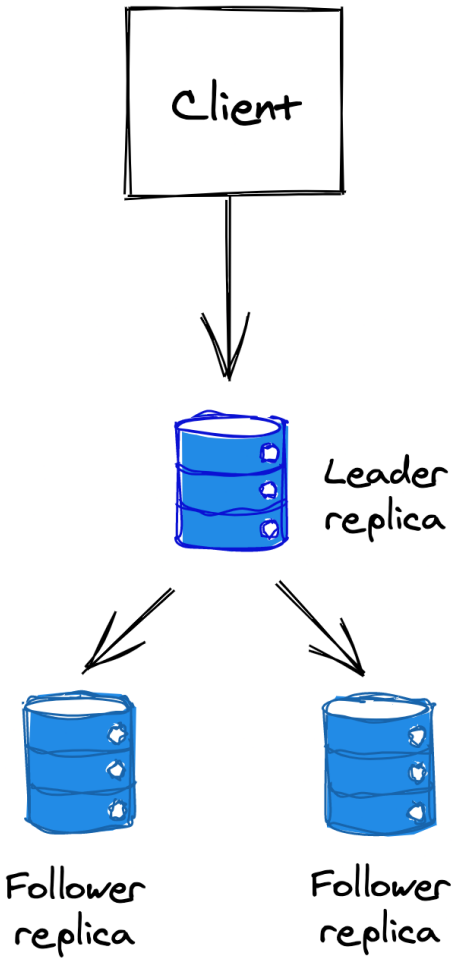


Figure 14.6: Single leader replication

At a high level, the replication can happen either fully synchronously, fully asynchronously, or as a combination of the two.

Asynchronous replication

In this mode, when the leader receives a write request from a client, it asynchronously sends out requests to the followers to replicate it and replies to the client before the replication has been completed.

Although this approach is fast, it's not fault-tolerant. What happens if the leader crashes right after accepting a write, but before replicating it to the followers? In this case, a new leader could be elected that doesn't have the latest updates, leading to data loss, which is one of the worst possible trade-offs you can make.

The other issue is consistency. A successful write might not be visible by some or all replicas because the replication happens asynchronously. The client could send a write to the leader and later fail to read the data from a replica because it doesn't exist there yet. The only guarantee is that if the writes stop, eventually, all replicas will catch up and be identical (eventual consistency).

Synchronous replication

Synchronous replication waits for a write to be replicated to all followers before returning a response to the client, which comes with a performance penalty. If a replica is extremely slow, every request will be affected by it. To the extreme, if any replica is down or not reachable, the store becomes unavailable and it can no longer write any data. The more nodes the data store has, the more likely a fault becomes.

As you can see, fully synchronous or asynchronous replication are extremes that provide some advantages at the expense of others. Most data stores have replication strategies that use a combination of the two. For example, in Raft, the leader replicates its writes to a majority before returning a

response to the client. And in PostgreSQL, you can configure a subset of replicas to receive updates synchronously rather than asynchronously.

14.2.2 Multi-leader replication

In multi-leader replication, there is more than one node that can accept writes. This approach is used when the write throughput is too high for a single node to handle, or when a leader needs to be available in multiple data centers to be geographically closer to its clients.

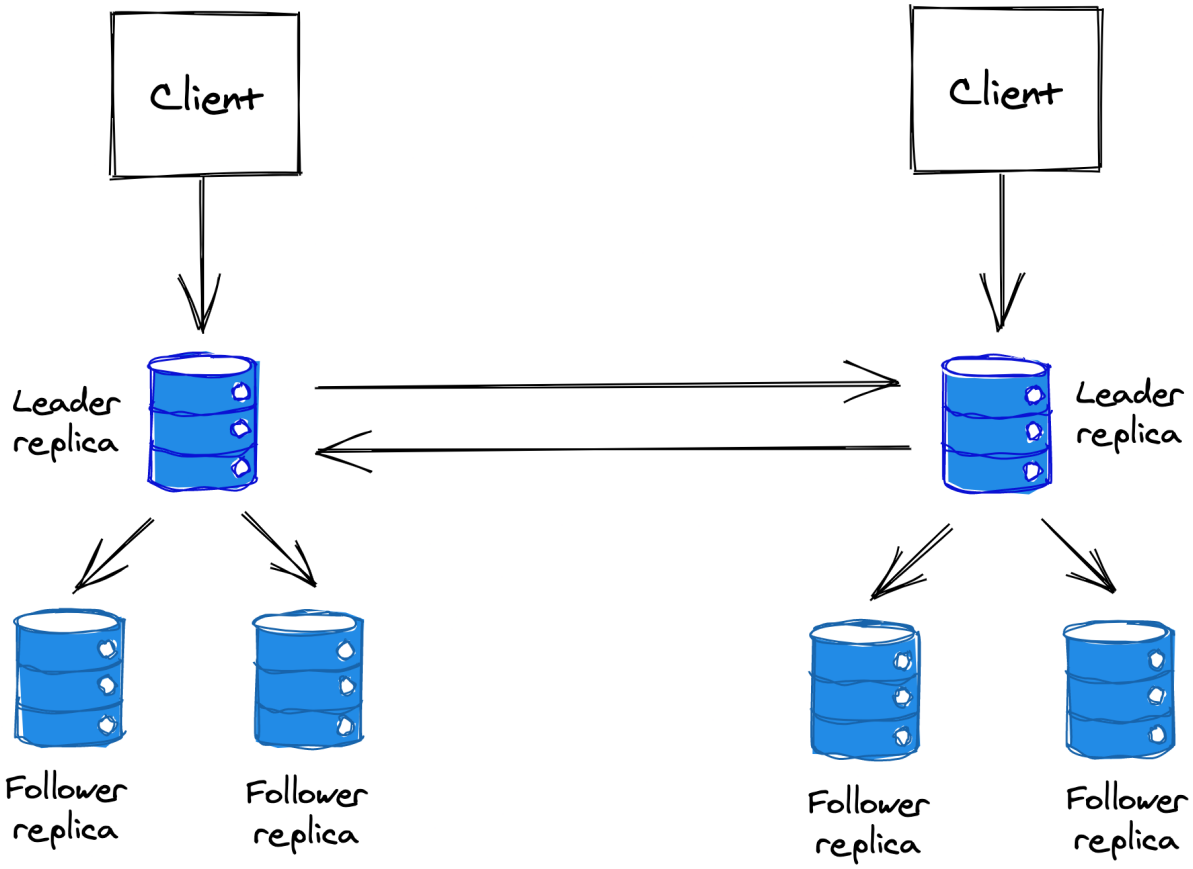


Figure 14.7: Multi-leader replication

The replication happens asynchronously since the alternative would defeat the purpose of using multiple leaders in the first place. This form of replication is generally best avoided when possible as it introduces a lot of complexity. The main issue with multiple leaders are conflicting writes; if the same data item is updated concurrently by two leaders, which one

should win? To resolve conflicts, the data store needs to implement a conflict resolution strategy.

The simplest strategy is to design the system so that conflicts are not possible; this can be achieved under some circumstances if the data has a homing region. For example, if all the European customer requests are always routed to the European data center, which has a single leader, there won't be any conflicting writes. There is still the possibility of a data center going down, but that can be mitigated with a backup data center in the same region, replicated with single-leader replication.

If assigning requests to specific leaders is not possible, and every client needs to be able to write to every leader, conflicting writes will inevitably happen.

One way to deal with a conflict updating a record is to store the concurrent writes and return them to the next client that reads the record. The client will try to resolve the conflict and update the data store with the resolution. In other words, the data store "pushes the can down the road" to the clients.

Alternatively, an automatic conflict resolution method needs to be implemented, for example:

- The data store could use the timestamps of the writes and let the most recent one win. This is generally not reliable because the nodes' physical clocks aren't perfectly synchronized. Logical clocks are better suited for the job in this case.
- The data store could allow the client to upload a custom conflict resolution procedure, which can be executed by the data store whenever a conflict is detected.
- Finally, the data store could leverage data structures that provide automatic conflict resolution, like a *conflict-free replicated data type* (CRDT). CRDTs are data structures that can be replicated across multiple nodes, allowing each replica to update its local version independently from others while resolving inconsistencies in a mathematically sound way.

14.2.3 Leaderless replication

What if any replica could accept writes from clients? In that case, there wouldn't be any leader(s), and the responsibility of replicating and resolving conflicts would be offloaded entirely to the clients.

For this to work, a basic invariant needs to be satisfied. Suppose the data store has N replicas. When a client sends a write request to the replicas, it waits for at least W replicas to acknowledge it before moving on. And when it reads an entry, it does so by querying R replicas and taking the most recent one from the response set. Now, as long as $W + R > N$, the write and replica set intersect, which guarantees that at least one record in the read set will reflect the latest write.

The writes are always sent to all N replicas in parallel; the W parameter determines just the number of responses the client has to receive to complete the request. The data store's read and write throughput depend on how large or small R and W are. For example, a workload with many reads benefits from a smaller R, but in turn, that makes writes slower and less available.

Like in multi-leader replication, a conflict resolution strategy needs to be used when two or more writes to the same record happen concurrently.

Leaderless replication is even more complex than multi-leader replication, as it's offloading the leader responsibilities to the clients, and there are edge cases that affect consistency even when $W + W > N$ is satisfied. For example, if a write succeeded on less than W replicas and failed on the others, the replicas are left in an inconsistent state.

14.3 Caching

Let's take a look now at a very specific type of replication that only offers best effort guarantees: caching.

Suppose a service requires retrieving data from a remote dependency, like a data store, to handle its requests. As the service scales out, the dependency needs to do the same to keep up with the ever-increasing load. A cache can

be introduced to reduce the load on the dependency and improve the performance of accessing the data.

A *cache* is a high-speed storage layer that temporarily buffers responses from downstream dependencies so that future requests can be served directly from it — it's a form of best effort replication. For a cache to be cost-effective, there should be a high probability that requested data can be found in it. This requires the data access pattern to have a high locality of reference, like a high likelihood of accessing the same data again and again over time.

14.3.1 Policies

When a cache miss occurs[3], the missing data item has to be requested from the remote dependency, and the cache has to be updated with it. This can happen in two ways:

- The client, after getting an "item-not-found" error from the cache, requests the data item from the dependency and updates the cache. In this case, the cache is said to be a *side cache*.
- Alternatively, if the cache is *inline*, the cache communicates directly with the dependency and requests the missing data item. In this case, the client only ever accesses the cache.

Because a cache has a maximum capacity for holding entries, an entry needs to be evicted to make room for a new one when its capacity is reached. Which entry to remove depends on the eviction policy used by the cache and the client's access pattern. One commonly used policy is to evict the least recently used (LRU) entry.

A cache also has an expiration policy that dictates for how long to store an entry. For example, a simple expiration policy defines the maximum time to live (TTL) in seconds. When a data item has been in the cache for longer than its TTL, it expires and can safely be evicted.

The expiration doesn't need to occur immediately, though, and it can be deferred to the next time the entry is requested. In fact, that might be

preferable — if the dependency is temporarily unavailable, and the cache is inline, it can opt to return an entry with an expired TTL to the client rather than an error.

14.3.2 In-process cache

The simplest possible cache you can build is an in-memory dictionary located within the clients, such as a hash-table with a limited size and bounded to the available memory that the node offers.

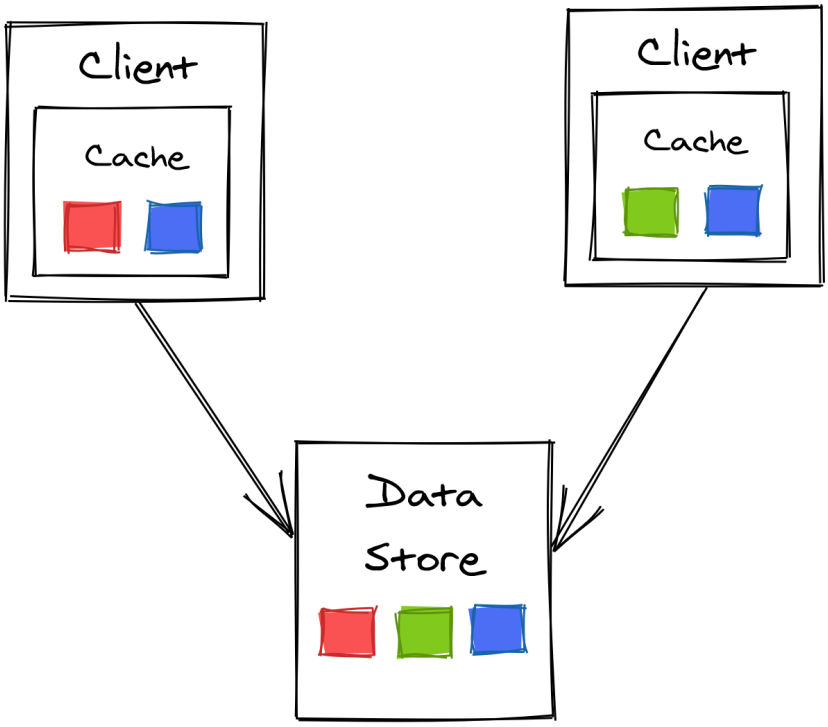


Figure 14.8: In-process cache

Because each cache is completely independent of the others, consistency issues are inevitable since each client potentially sees a different version of the same entry. Additionally, an entry needs to be fetched once per cache, creating downstream pressure proportional to the number of clients.

This issue is exacerbated when a service with an in-process cache is restarted or scales out, and every newly started instance requires to fetch entries directly from the dependency. This can cause a “thundering herd”

effect where the downstream dependency is hit with a spike of requests. The same can happen at run-time if a specific data item that wasn't accessed before becomes all of a sudden very popular.

Request coalescing can be used to reduce the impact of a thundering herd. The idea is that there should be at most one outstanding request at the time to fetch a specific data item per in-process cache. For example, if a service instance is serving 10 concurrent requests requiring a specific record that is not yet in the cache, the instance will send only a single request out to the remote dependency to fetch the missing entry.

14.3.3 Out-of-process cache

An external cache, shared across all service instances, addresses some of the drawbacks of using an in-process cache at the expense of greater complexity and cost.

Because the external cache is shared among the service instances, there can be only a single version of each data item at any given time. And although the cached item can be out-of-date, every client accessing the cache will see the same version, which reduces consistency issues. The load on the dependency is also reduced since the number of times an entry is accessed no longer grows as the number of clients increases.

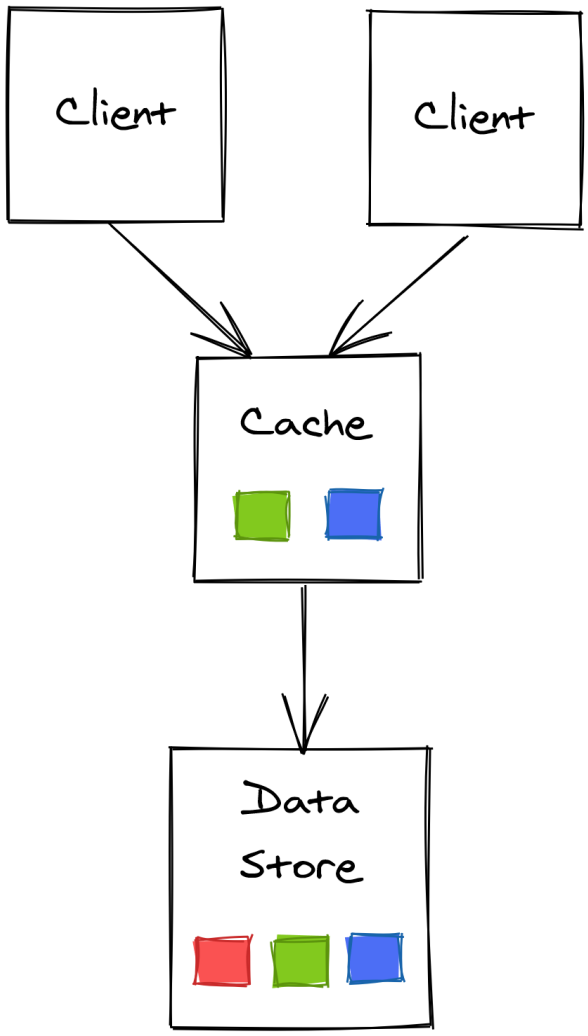


Figure 14.9: Out-of-process cache

Although we have managed to decouple the clients from the dependency, we have merely shifted the load to the external cache. If the load increases, the cache will eventually need to be scaled out. As little data as possible should be moved around when that happens to guarantee that the cache's availability doesn't drop and that the number of cache misses doesn't significantly increase. Consistent hashing, or a similar partitioning technique, can be used to reduce the amount of data that needs to be moved around.

Maintaining an external cache comes with a price as it's yet another service that needs to be maintained and operated. Additionally, the latency to access

it is higher than accessing an in-process cache because a network call is required.

If the external cache is down, how should the service react? You would think it might be okay to temporarily bypass the cache and directly hit the dependency. But in that case, the dependency might not be prepared to withstand a surge of traffic since it's usually shielded by the cache. Consequently, the external cache becoming unavailable could easily cause a cascading failure resulting in the dependency to become unavailable as well.

The clients can leverage an in-process cache as a defense against the external cache becoming unavailable. Similarly, the dependency also needs to be prepared to handle these sudden "attacks." Load shedding is a technique that can be used here, which we will discuss later in the book.

What's important to understand is that a cache introduces a bi-modal behavior in the system[4]. Most of the time, the cache is working as expected, and everything is fine; when it's not for whatever reason, the system needs to survive without it. It's a design smell if your system can't cope at all without a cache.

---

1. This is also referred to as layer 4 (L4) load balancing since layer 4 is the transport layer in the OSI model.↩

2. Also referred to as a layer 7 (L7) load balancer since layer 7 is the application layer in the OSI model↩

3. A cache hit occurs when the requested data can be found in the cache, while a cache miss occurs when it cannot.↩

4. Remember when we talked about the bi-modal behavior of message channels in section 12.4? As we will learn later, you always want to minimize the number of modes in your applications to make them simple to understand and operate.↩

(PART) Resiliency

Introduction

As you scale out your applications, any failure that can happen will eventually happen. Hardware failures, software crashes, memory leaks — you name it. The more components you have, the more failures you will experience.

Suppose you have a buggy service that leaks 1 MB of memory on average every hundred requests. If the service does a thousand requests per day, chances are you will restart the service to deploy a new build before the leak reaches any significant size. But if your service is doing 10 million requests per day, then by the end of the day you lose 100 GB of memory! Eventually, the servers won't have enough memory available and they will start to trash due to the constant swapping of pages in and out from disk.

This nasty behavior is caused by *cruel math*; given an operation that has a certain probability of failing, the total number of failures increases with the total number of operations performed. In other words, the more you scale out your system to handle more load, and the more operations and moving parts there are, the more failures your systems will experience.

Remember when we talked about availability and "nines" in chapter 1? Well, to guarantee just two nines, your system can be unavailable for up to 15 min a day. That's very little time to take any manual action. If you strive for 3 nines, then you only have 43 minutes per *month* available. Although you can't escape cruel math, you can mitigate it by implementing self-healing mechanisms to reduce the impact of failures.

Chapter 15 describes the causes of the most common failures in distributed systems: single points of failure, unreliable networks, slow processes, and unexpected load.

Chapter 16 dives into resiliency patterns that help shield a service against failures in downstream dependencies, like timeouts, retries, and circuit breakers.

Chapter 17 discusses resiliency patterns that help protect a service against upstream pressure, like load shedding, load leveling, and rate-limiting.

15 Common failure causes

In order to protect your systems against failures, you first need to have an idea of what can go wrong. The most common failures you will encounter are caused by single points of failure, the network being unreliable, slow processes, and unexpected load. Let's take a closer look at these.

15.1 Single point of failure

A single point of failure is the most glaring cause of failure in a distributed system; if it were to fail, that one component would bring down the entire system with it. In practice, systems can have multiple single points of failure.

A service that starts up by needing to read a configuration from a non-replicated database is an example of a single point of failure; if the database isn't reachable, the service won't be able to (re)start. A more subtle example is a service that exposes an HTTP API on top of TLS using a certificate that needs to be manually renewed. If the certificate isn't renewed by the time it expires, then all clients trying to connect to it wouldn't be able to open a connection with the service.

Single points of failure should be identified when the system is architected before they can cause any harm. The best way to detect them is to examine every component of the system and ask what would happen if that component were to fail. Some single points of failure can be architected away, e.g., by introducing redundancy, while others can't. In that case, the only option left is to minimize the blast radius.

15.2 Unreliable network

When a client makes a remote network call, it sends a request to a server and expects to receive a response from it a while later. In the best case, the client receives a response shortly after sending the request. But what if the

client waits and waits and still doesn't get a response? In that case, the client doesn't know whether a response will eventually arrive or not. At that point it has only two options: it can either continue to wait, or fail the request with an exception or error.

As discussed when the concept of failure detection was introduced in chapter 7, there are several reasons why the client hasn't received a response so far:

- the server is slow;
- the client's request has been dropped by a network switch, router or proxy;
- the server has crashed while processing the request;
- or the server's response has been dropped by a network switch, router or proxy.

Slow network calls are the silent killers of distributed systems. Because the client doesn't know whether the response is on its way or not, it can spend a long time waiting before giving up, if it gives up at all. The wait can in turn cause degradations that are extremely hard to debug. In chapter 16 we will explore ways to protect clients from the unreliability of the network.

15.3 Slow processes

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. Whenever you use resources, especially when they have been leased from a pool, there is a potential for leaks.

Memory is the most well-known source of leaks. A memory leak causes a steady increase in memory consumption over time. Run-times with garbage collection don't help much either; if a reference to an object that is no longer needed is kept somewhere, the object won't be deleted by the garbage collector.

A memory leak keeps consuming memory until there is no more of it, at which point the operating system starts swapping memory pages to disk constantly, while the garbage collector kicks in more frequently trying its best to release any shred of memory. The constant paging and the garbage collector eating up CPU cycles make the process slower. Eventually, when there is no more physical memory, and there is no more space in the swap file, the process won't be able to allocate more memory, and most operations will fail.

Memory is just one of the many resources that can leak. For example, if you are using a thread pool, you can lose a thread when it blocks on a synchronous call that never returns. If a thread makes a synchronous blocking HTTP call without setting a timeout, and the call never returns, the thread won't be returned to the pool. Since the pool has a fixed size and keeps losing threads, it will eventually run out of threads.

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 pay a hefty 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 size, eventually there won't be any connections left.

On top of that, the code you write isn't the only one accessing memory, threads, and sockets. The libraries your application depends on access the same resources, and they can do all kinds of shady things. Without digging into their implementation, assuming it's open in the first place, you can't be sure whether they can wreak havoc or not.

15.4 Unexpected load

Every system has a limit to how much load it can withstand without scaling. Depending on how the load increases, you are bound to hit that brick wall sooner or later. One thing is an organic increase in load that gives you the time to scale out your service accordingly, but another is a sudden and unexpected spike.

For example, consider the number of requests received by a service 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 — depending on the hour of the day, the service is going to get hit by users in different countries.
- Some requests are much more expensive than others and abuse the system in ways you might have not anticipated, like scrapers slurping in data from your site at super-human speed.
- Some requests are malicious, like DDoS attacks that try to saturate your service's bandwidth, denying access to the service for legitimate users.

To withstand unexpected load, you need to prepare beforehand. The patterns in chapter 17 will teach you some techniques on how to do just that[1].

15.5 Cascading failures

You would think that if your system has hundreds of processes, it shouldn't make much difference if a small percentage are slow or unreachable. The thing about failures is that they tend to spread like cancer, propagating from one process to the other until the whole system crumbles to its knees. This effect is also referred to as a *cascading failure*, which occurs when a portion of an overall system fails, increasing the probability that other portions fail.

For example, suppose there are multiple clients querying two database replicas A and B, which are behind a load balancer. Each replica is handling about 50 transactions per second (see Figure 15.1).

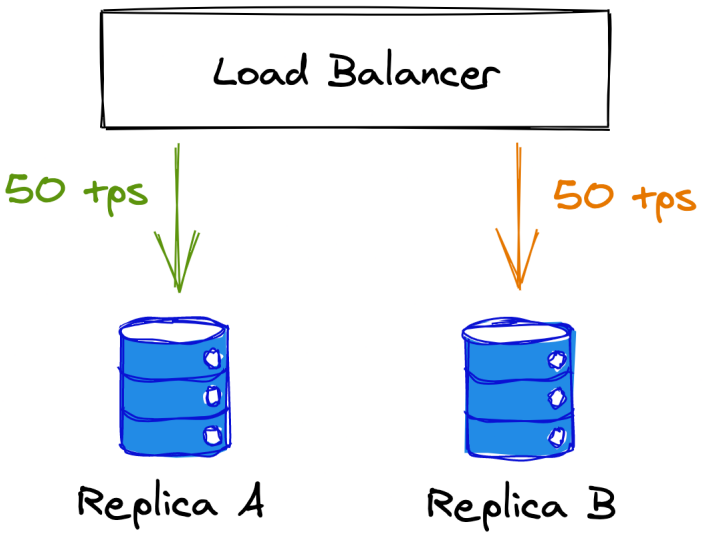


Figure 15.1: Two replicas behind an LB; each is handling half the load.

Suddenly, replica B becomes unavailable because of a network fault. The load balancer detects that B is unavailable and removes it from its pool. Because of that, replica A has to pick up the slack for replica B, doubling the load it was previously under (see Figure 15.2).

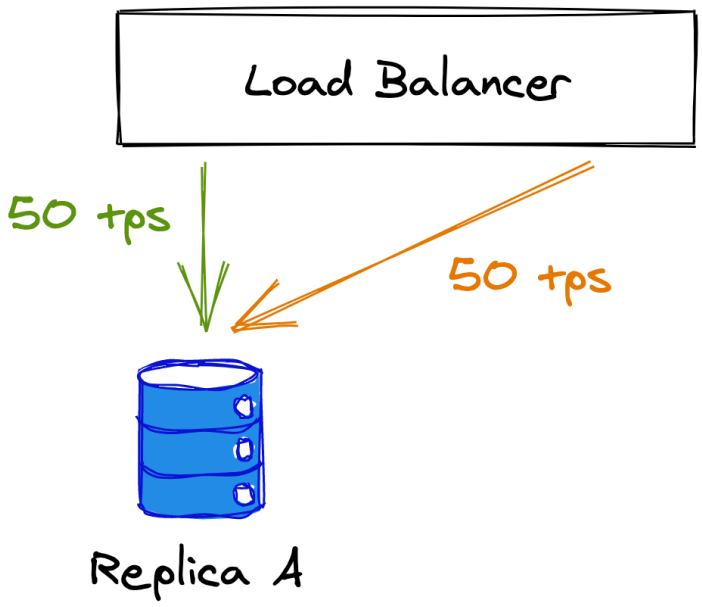


Figure 15.2: When replica B becomes unavailable, A will be hit with more load, which can strain it beyond its capacity.

As replica A starts to struggle to keep up with the incoming requests, the clients experience more failures and timeouts. In turn, they retry the same failing requests several times, adding insult to injury.

Eventually, replica A is under so much load that it can no longer serve requests promptly and becomes unavailable, causing replica A to be removed from the load balancer's pool. In the meantime, replica B becomes available again and the load balancer puts it back in the pool, at which point it's flooded with requests that kill the replica instantaneously. This feedback loop of doom can repeat several times.

Cascading failures are very hard to get under control once they have started. The best way to mitigate one is to not have it in the first place. The patterns introduced in the next chapters will help you stop the cracks in the system from spreading.

15.6 Risk management

As we have just seen, a distributed system needs to embrace that failures will happen and needs to be prepared for it. Just because a failure has a chance of happening doesn't always mean you have necessarily to do something about it. The day has only so many hours, and you will need to make tough decisions about where to spend your engineering time.

Given a specific failure, you have to consider its probability of happening and the impact it causes to your system if it does happen. By multiplying the two factors together, you get a risk score, which you can then use to decide which failures to prioritize and act upon (see Figure 15.3). A failure that is very likely to happen, and has an extensive impact, should be dealt with swiftly; on the other hand, a failure with a low likelihood and low impact can wait.

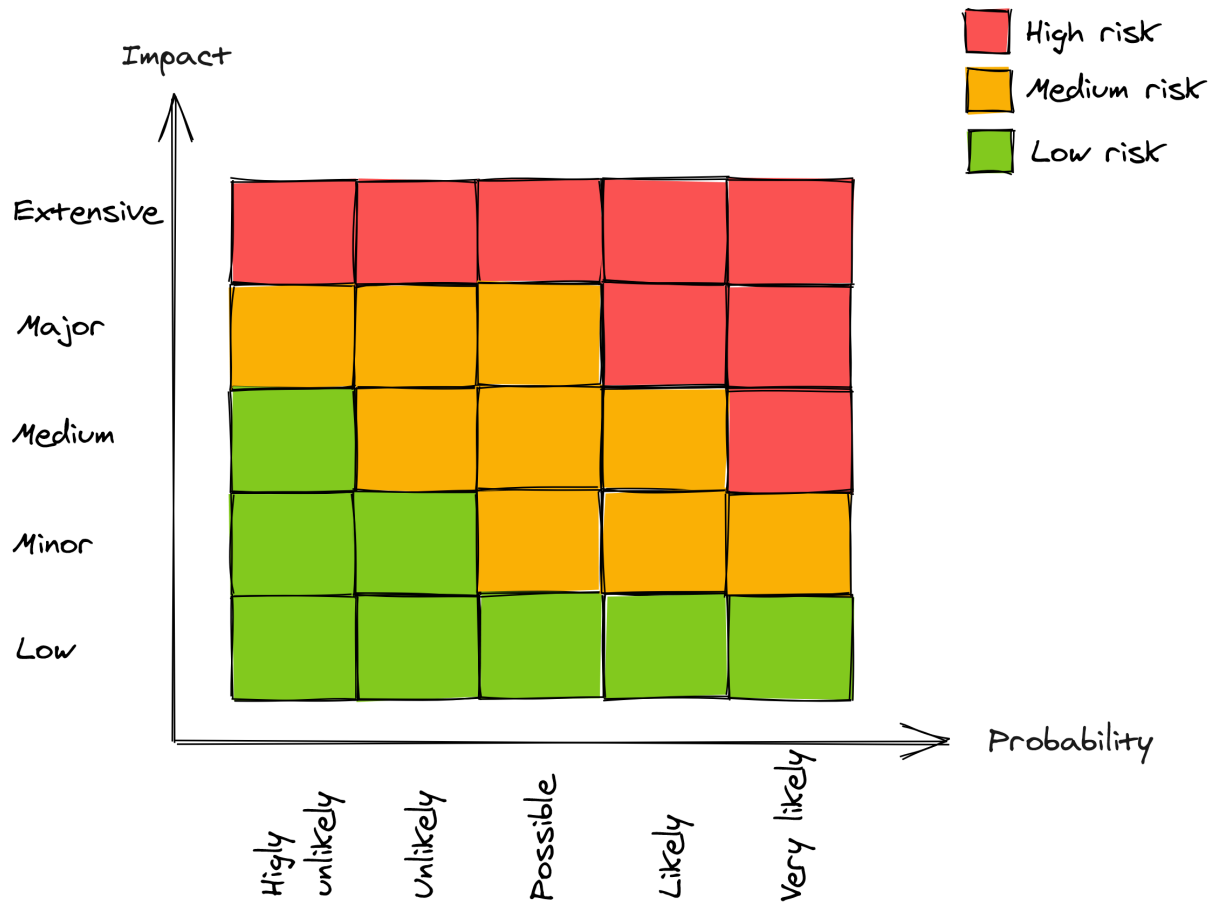


Figure 15.3: Risk matrix

To address a failure, you can either find a way to reduce the probability of it happening, or reduce its impact.

---

1. These techniques might look simple but are very effective. During the COVID-19 outbreak, I have witnessed many of the systems I was responsible for at the time doubling traffic nearly overnight without causing any incidents.↩

16 Downstream resiliency

In this chapter, we will explore patterns that shield a service against failures in its downstream dependencies.

16.1 Timeout

When you make a network call, you can configure a timeout to fail the request if there is no response within a certain amount of time. If you make the call without setting a timeout, you tell your code that you are 100% confident that the call will succeed. Would you really take that bet?

Unfortunately, some network APIs don't have a way to set a timeout in the first place. When the default timeout is infinity, it's all too easy for a client to shoot itself in the foot. As mentioned earlier, network calls that don't return lead to resource leaks at best. Timeouts limit and isolate failures, stopping them from cascading to the rest of the system. And they are useful not just for network calls, but also for requesting a resource from a pool and for synchronization primitives like mutexes.

To drive the point home on the importance of setting timeouts, let's take a look at some concrete examples. JavaScript's *XMLHttpRequest* is *the* web API to retrieve data from a server asynchronously. Its default timeout is zero, which means there is no timeout:

```
var xhr = new XMLHttpRequest();
xhr.open('GET', '/api', true);
// No timeout by default!
xhr.timeout = 10000;
xhr.onload = function () {
 // Request finished
};
xhr.ontimeout = function (e) {
 // Request timed out
};
xhr.send(null);
```

Client-side timeouts are as crucial as server-side ones. There is a maximum number of sockets your browser can open for a particular host. If you make network requests that never return, you are going to exhaust the socket pool. When the pool is exhausted, you are no longer able to connect to the host.

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. Browsers have recently added experimental support for the Abort API to support timeouts.

```
const controller = new AbortController();
const signal = controller.signal;
const fetchPromise = fetch(url, {signal});
// No timeout by default!
setTimeout(() => controller.abort(), 10000);
fetchPromise.then(response => {
 // Request finished
})
```

Things aren't much rosier for Python. The popular *requests* library uses a default timeout of infinity:

```
# No timeout by default!
response = requests.get('https://github.com/', timeout=10)
```

Go's *HTTP package* doesn't use timeouts by default, either:

```
var client = &http.Client{
  // No timeout by default!
  Timeout: time.Second * 10,
}
response, _ := client .Get(url)
```

Modern HTTP clients for Java and .NET do a much better job and usually come with default timeouts. For example, .NET Core *HttpClient* has a default timeout of 100 seconds. It's lax but better than not setting a timeout at all.

As a rule of thumb, always set timeouts when making network calls, and be wary of third-party libraries that do network calls or use internal resource pools but don't expose settings for timeouts. And if you build libraries, always set reasonable default timeouts and make them configurable for your clients.

Ideally, you should set your timeouts based on the desired false timeout rate. Say you want to have about 0.1% false timeouts; to achieve that, you should set the timeout to the 99.9th percentile of the remote call's response time, which you can measure empirically.

You also want to have good monitoring in place to measure the entire lifecycle of your network calls, like the duration of the call, the status code received, and if a timeout was triggered. We will talk about monitoring later in the book, but the point I want to make here is that you have to measure what happens at the integration points of your systems, or you won't be able to debug production issues when they show up.

Ideally, you want to encapsulate a remote call within a library that sets timeouts and monitors it for you so that you don't have to remember to do this every time you make a network call. No matter which language you use, there is likely a library out there that implements some of the resiliency and transient fault-handling patterns introduced in this chapter, which you can use to encapsulate your system's network calls.

Using a language-specific library is not the only way to wrap your network calls; you can also leverage a reverse proxy co-located on the same machine which intercepts all the remote calls that your process makes[1]. The proxy enforces timeouts and also monitors the calls, relinquishing your process from the responsibility to do so.

16.2 Retry

You 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 the timeout fires? The client has two options at that point: it can either fail fast or retry the request at a later time.

If the failure or timeout was caused by a short-lived connectivity issue, then retrying after some *backoff time* has a high probability of succeeding. However, if the downstream service is overwhelmed, retrying immediately will only make matters worse. 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 a certain amount of time has passed since the initial request.

16.2.1 Exponential backoff

To set the delay between retries, you can use a *capped exponential function*, where the delay is derived by multiplying the initial backoff duration by a constant 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, there is still a problem. When the downstream service is temporarily degraded, it's likely that multiple clients see their requests failing around the same time. This causes the clients to retry simultaneously, hitting the downstream service with load spikes that can further degrade it, as shown in Figure 16.1.

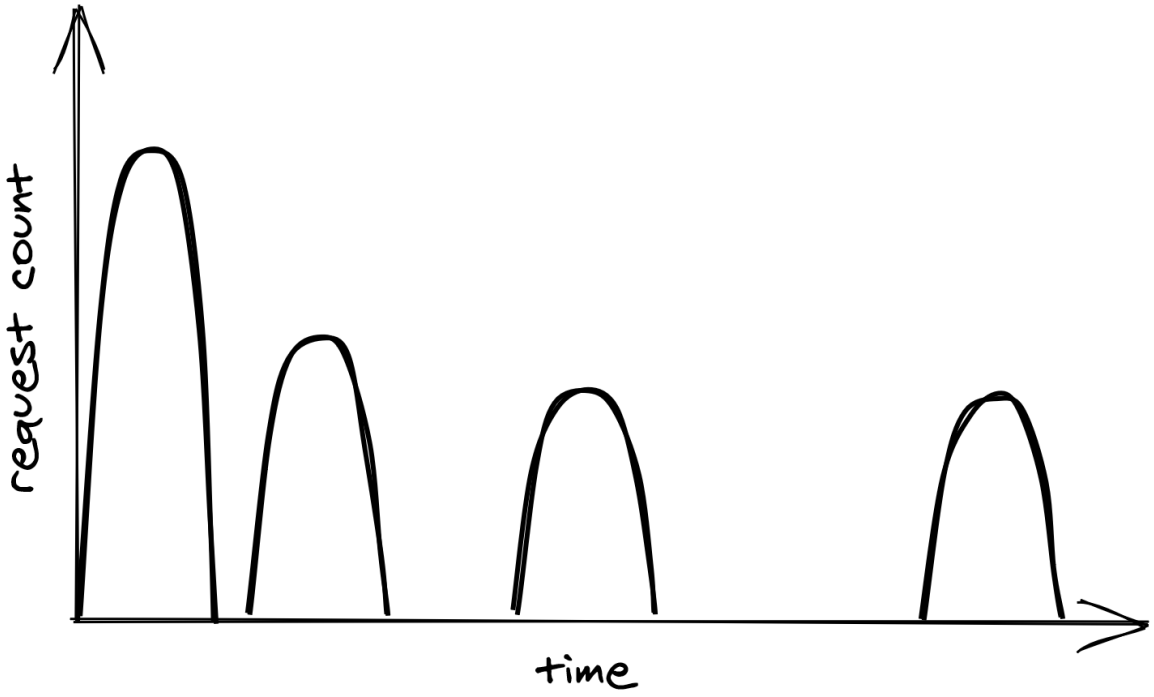


Figure 16.1: Retry storm

To avoid this herding behavior, you can introduce random jitter in the delay calculation. With it, the retries spread 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 failed requests into a *retry queue*. The same process, or possibly another, reads from the same queue later and retries the 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, then 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.

You should also not retry a network call that isn't idempotent, and whose side effects can affect your application's correctness. Suppose a process is making a call to a payment provider service, and the call times out; should it retry or not? The operation might have succeeded and retrying would charge the account twice, unless the request is idempotent.

16.2.2 Retry amplification

Suppose that handling a request from a client requires it to go through a chain of dependencies. The client makes a call to service A, which to handle the request talks to service B, which in turn talks to 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, which in turn makes it more likely to hit A's timeout. If that happens, A retries its request again, 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 number of retries; the deeper a service is in the chain, the higher the load it will be exposed to due to the amplification (see Figure 16.2).

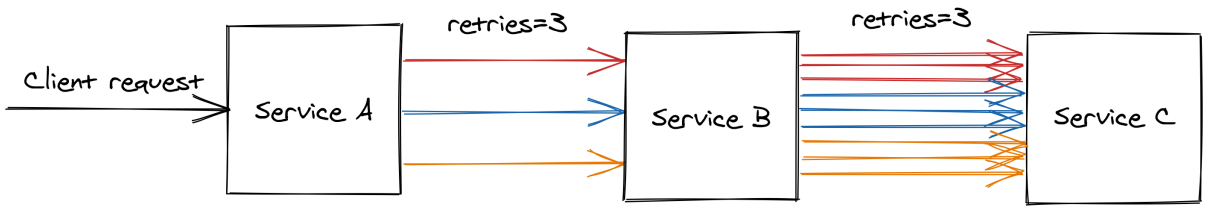


Figure 16.2: Retry amplification in action

And if the pressure gets bad enough, this behavior can easily bring down the whole system. That's why when you have long dependency chains, you should only retry at a single level of the chain, and fail fast in all the other ones.

16.3 Circuit breaker

Suppose your service uses timeouts to detect communication failures with a downstream dependency, and retries to mitigate transient failures. If the failures aren't transient and the downstream dependency keeps being 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 propagate to the rest of the system and cause cascading failures.

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 you don't have to make. This mechanism is also called a circuit breaker, inspired by the same functionality implemented in electrical circuits.

A circuit breaker's goal is to allow a sub-system to fail without bringing down the whole system with it. 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, which makes the pattern particularly useful for long-term degradations. 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.

16.3.1 State machine

The circuit breaker is implemented as a state machine that can be in one of three states: open, closed and half-open (see Figure 16.3).

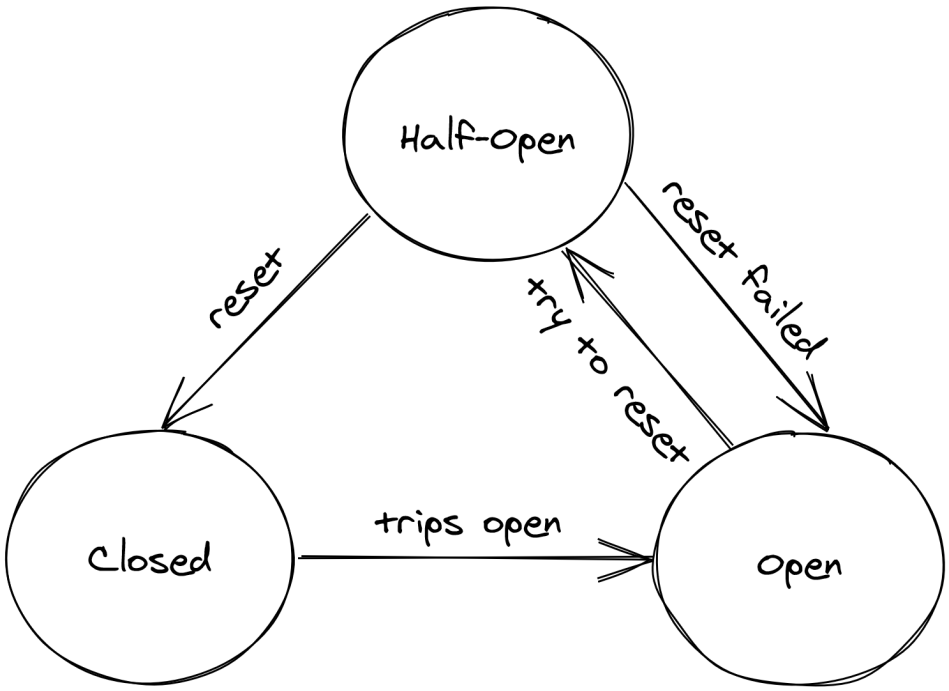


Figure 16.3: Circuit breaker state machine

In the closed state, the circuit breaker is merely acting 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, you need to think carefully what should happen when a downstream dependency is down. If the down-stream dependency is non-critical, you want your 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 not available, the page should render without recommendations. It's a better outcome than to fail the rendering of the whole page entirely.

After some time has passed, the circuit breaker decides to give 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.

That's really all there is to understand how a circuit breaker works, but the devil is in the details. 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 your specific case; only by using data about past failures can you make an informed decision.

---

1. We talked about this in section 14.1.3 when discussing the sidecar pattern and the service mesh.↩

17 Upstream resiliency

So far, we have discussed patterns that protect 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.

17.1 Load shedding

A server has very little control over how many requests it receives at any given time, which can deeply impact its performance.

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 starves out of resources like memory, threads, sockets, or files. This causes the response time to increase to the point the server becomes unavailable to the outside world.

When a server operates at capacity, there is no good reason for it to keep accepting new requests since that will only end up degrading it. In that case, the process should start rejecting excess requests so that it can focus on the ones it is already processing.

The definition of overload depends on your system, but the general idea is that it should be measurable and actionable. For example, the number of concurrent requests being processed is a good candidate to measure a server's load; all you have to do is to increment a counter when a new request comes in and decrease it when the server has processed it and sent back a response to the client.

When the server detects that it's overloaded, it can reject incoming requests by failing fast and returning a *503 (Service Unavailable)* status code in the response. This technique is also referred to as load shedding. The server doesn't necessarily have to reject arbitrary requests though; for example, if

different requests have different priorities, the server could reject only the lower-priority ones.

Unfortunately, rejecting a request doesn't completely offload from the server 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 read the request just to finally reject it. Hence, load shedding can only help so much, and if the load keeps increasing, eventually, the cost of rejecting requests takes over, and the service starts to degrade.

17.2 Load leveling

Load leveling is an alternative to load shedding, which can be used when clients don't expect a response within a short time frame.

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 the service to process requests at its own pace — rather than requests being pushed to the service by the clients, they are pulled by the service from the channel. This pattern is referred to as load leveling and it's well suited to fend off short-lived spikes, which the channel smoothes out (see Figure 17.1).

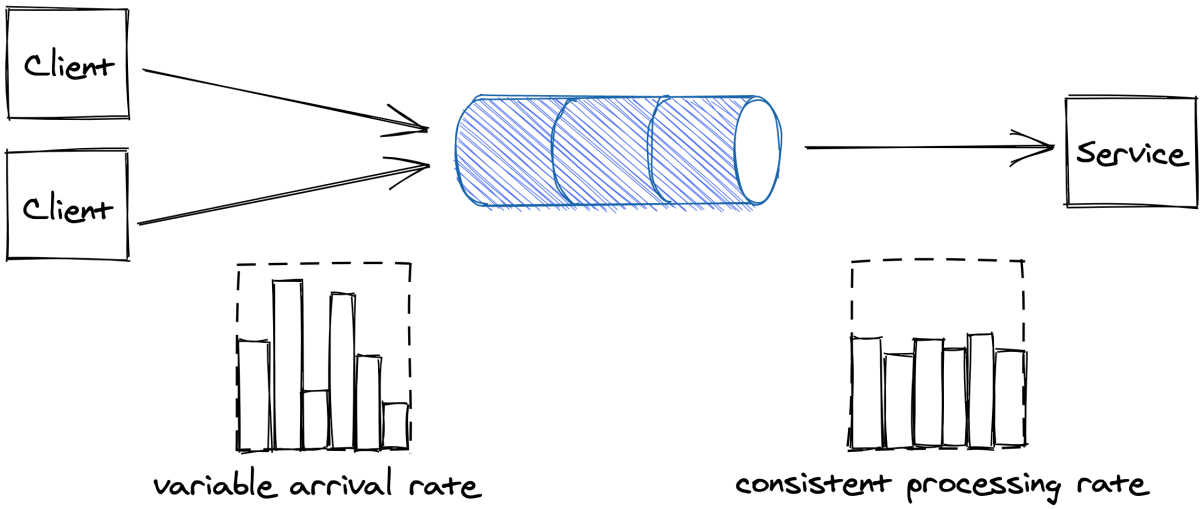


Figure 17.1: The channel smooths out the load for the consuming service.

Load-shedding and load leveling don't address an increase in load 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, which detects that the service is running hot and automatically increases its scale to handle the additional load.

17.3 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, like for the number of requests seen, or the number of 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 tagged with 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 breached. For services with HTTP APIs, 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 breached and by how much; it can also include a *Retry-After* header indicating how long to wait before making a new request:

```
HTTP/1.1 429 Too Many Requests
Retry-After: 60
```

If the client application plays by the rules, it stops hammering the service for some time, protecting it from non-malicious users monopolizing it by mistake. This protects against bugs in the clients that, for one reason or another, cause a client to repeatedly hit a downstream service for no good reason.

Rate-limiting is also used to enforce pricing tiers; if a user wants to use more resources, they also need to be prepared 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 denial-of-service (DDoS) attack, but it only partially protects a service from it. Nothing forbids throttled clients from continuing to hammer a service after getting *429s*. And no, rate-limited requests aren't free either — for example, to rate-limit a request by API key, the service has to pay the price to open a TLS connection, and to 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 frontend service, no matter which of the services behind it are attacked, the frontend 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 frontend service is amortized across all the services that are using it.

Although rate-limiting has some similarities to 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.

17.3.1 Single-process implementation

The implementation of rate-limiting is interesting in its own right, and it's well worth spending some time studying it, as a similar approach can be applied to other use cases. We will start with single-process implementation first and then proceed with 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. Every time 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-limits incoming requests by comparing it with the quota. The problem with this approach is that it requires a list per API key, which becomes quickly 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 compress the storage required. One way to do this is to divide time into buckets of fixed time duration, for example of 1 minute, and keep track of how many requests have been seen within each bucket (see Figure 17.2).

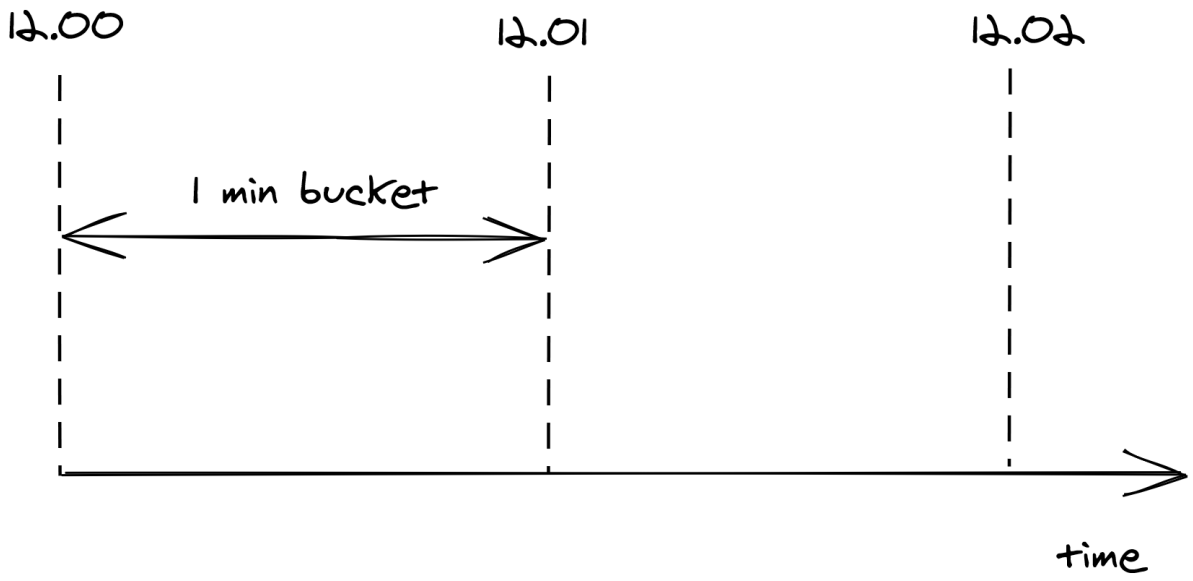


Figure 17.2: Buckets divide time into 1-minute intervals, which keep track of the number of requests seen.

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 17.3).

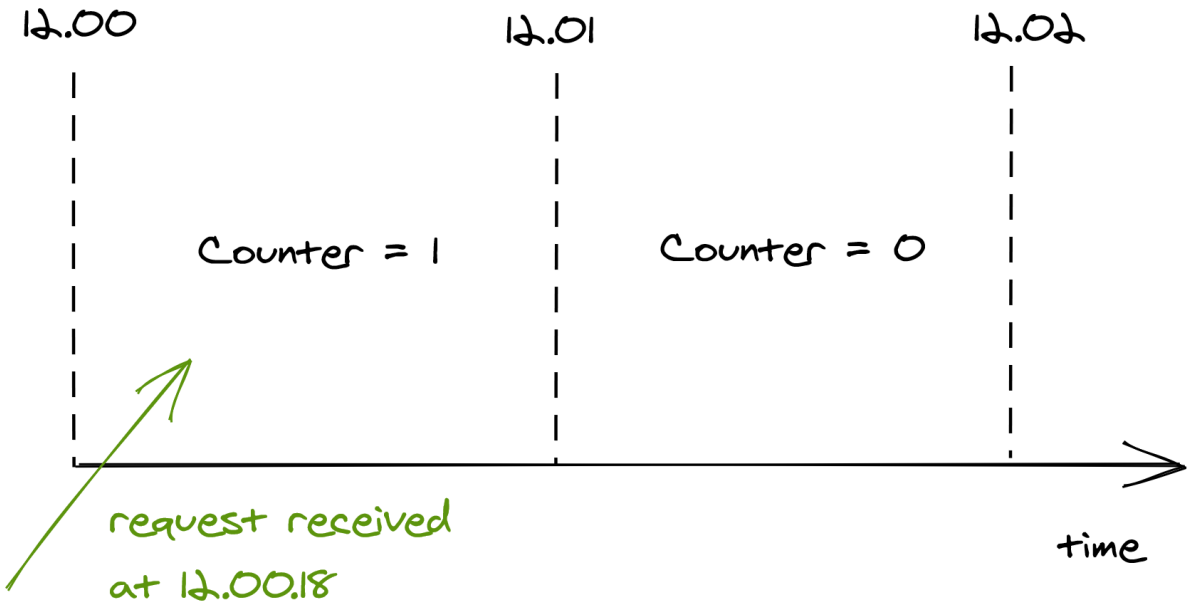


Figure 17.3: When a new request comes in, its timestamp is used to determine the bucket it belongs to.

With bucketing, we can compress the information about the number of requests seen in a way that doesn't grow as the number of requests does. Now that we have a memory-friendly representation, how can we use it to implement rate-limiting? The idea is to use a sliding window that moves in real-time across the buckets, 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 17.4).

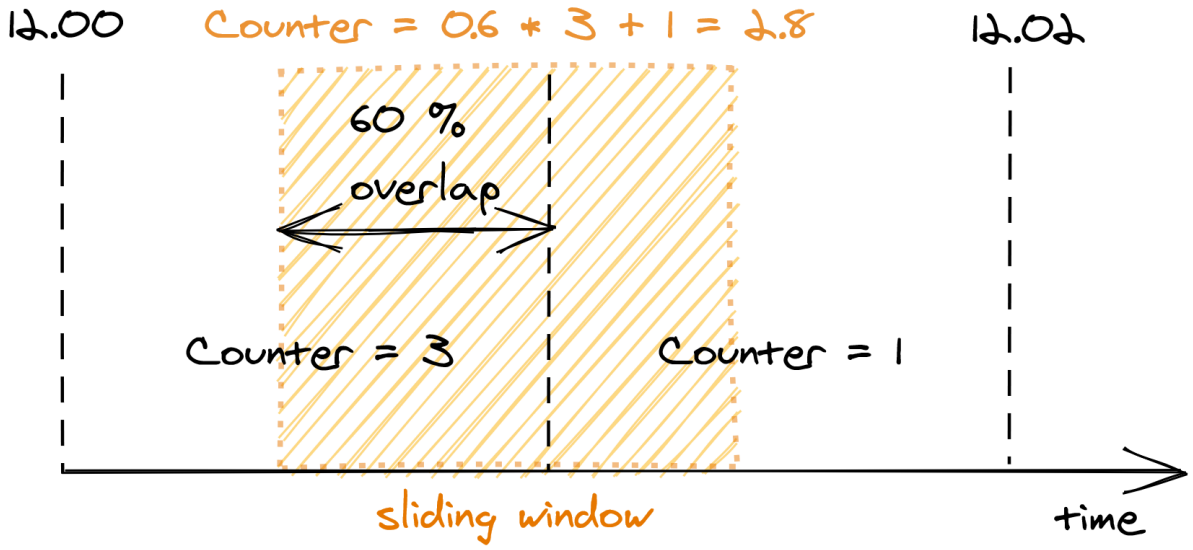


Figure 17.4: A bucket's weight is proportional to its overlap with the sliding window.

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. For example, you 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 touch at most 2 buckets. And if it can touch at most two buckets, there is no point to store the third oldest bucket, the fourth oldest one, and so on.

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.

17.3.2 Distributed implementation

When more than one process accepts requests, the local state no longer cuts it, 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 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. To avoid any race conditions, the fetch, update, and write operations need to be packaged into a single transaction.

Although this approach is functionally correct, it's costly. There are two issues here: transactions are slow, and executing one per request would be crazy expensive as the database would have to scale linearly with the number of requests. On top of that, for each request a process receives, it needs to do an outgoing call to a remote data store. What should it do if it fails?

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*. Atomic operations have much better performance than transactions.

Now, rather than updating the database on each request, the process can batch bucket updates in memory for some time, and flush them asynchronously to the database at the end of it (see Figure 17.5). This reduces the shared state's accuracy, but it's a good trade-off as it reduces the load on the database and the number of requests sent to it.

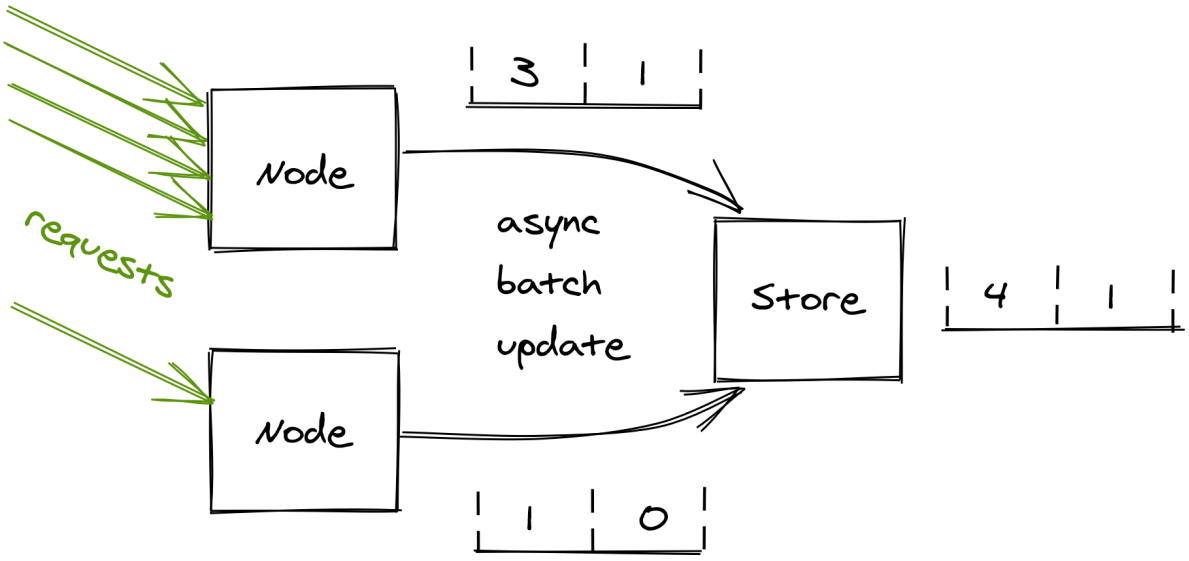


Figure 17.5: Servers batch bucket updates in memory for some time, and flush them asynchronously to the database at the end of it.

What happens if the database 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 all incoming requests just because the database used for rate-limiting is not reachable could be very damaging to the business. Instead, it's safer to keep serving requests based on the last state read from the store.

17.4 Bulkhead

The goal of the bulkhead pattern is to isolate a fault in one part of a service from taking the entire service down with it. The pattern is named after the partitions of a ship's hull. If one partition is damaged and fills up with water, the leak is isolated to that partition and doesn't spread to the rest of the ship.

Some clients can create much more load on a service than others. Without any protections, a single greedy client can hammer the system and degrade every other client. We have seen some patterns, like rate-limiting, that help prevent a single client from using more resources than it should. But rate-

limiting is not bulletproof. You can rate-limit clients based on the number of requests per second; but what if a client sends very heavy or poisonous requests that cause the servers to degrade? In that case, rate-limiting wouldn't help much as the issue is intrinsic with the requests sent by that client, which could eventually lead to degrading the service for every other client.

When everything else fails, the bulkhead pattern provides guaranteed fault isolation by design. The idea is to partition a shared resource, like a pool of service instances behind a load balancer, and assign each user of the service to a specific partition so that its requests can only utilize resources belonging to the partition it's assigned to.

Consequently, a heavy or poisonous user can only degrade the requests of users within the same partition. For example, suppose there are 10 instances of a service behind a load balancer, which are divided into 5 partitions (see Figure 17.6). In that case, a problematic user can only ever impact 20 percent of the service's instances. The problem is that the unlucky users who happen to be on the same partition as the problematic one are fully impacted. Can we do better?

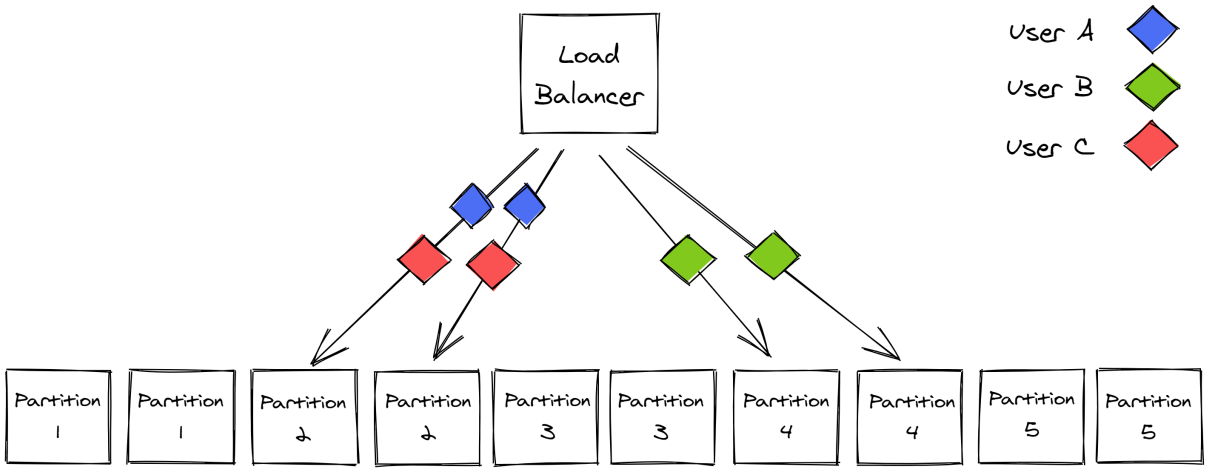


Figure 17.6: Service instances partitioned into 5 partitions

We can introduce *virtual partitions* that are composed of a random subset of instances. This can make it much more unlikely for another user to be allocated to the exact same virtual partition.

In our example, we can extract 45 combinations of 2 instances (virtual partitions) from a pool of 10 instances. When a virtual partition is degraded, other virtual partitions are only partially impacted as they don't fully overlap (see Figure 17.7). If you combine this with a health check on the load balancer, and a retry mechanism on the client side, what you get is much better fault isolation.

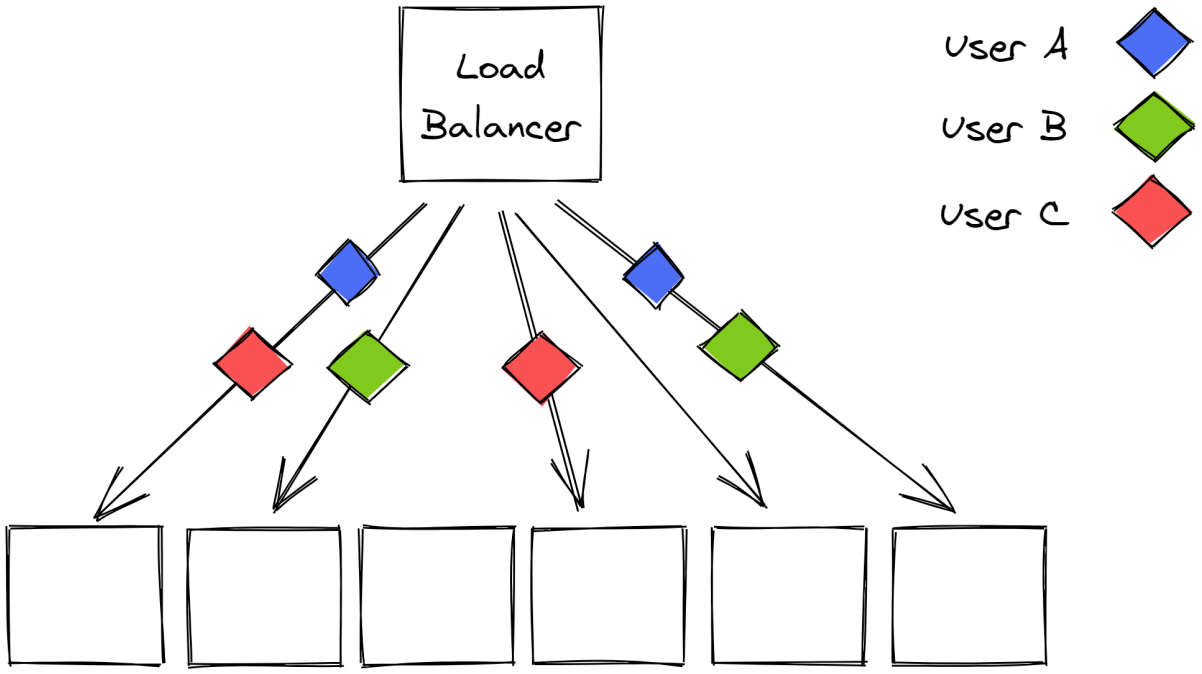


Figure 17.7: Virtual partitions are far less likely to fully overlap with each other.

You need to be careful when applying the bulkhead pattern; if you take it too far and create too many partitions, you lose all the economy-of-scale benefits of sharing costly resources across a set of users that are active at different times.

You also introduce a scaling problem. Scaling is simple when there are no partitions and every user can be served by any instance, as you can just add more instances. It's not that easy with a partitioned pool of instances as some partitions are much hotter than others.

17.5 Health endpoint

So far, we have explored patterns that allow a process to shed or reject incoming requests. Those are mitigations a server can apply only after it has received a request. Wouldn't it be nice to have a way to control the incoming traffic so that it doesn't reach a degraded server in the first place?

If the server is behind a load balancer and can communicate that it's overloaded, the balancer can stop sending requests to it. The process can expose a health endpoint that when queried performs a health check that either returns *200 (OK)* if the process can serve requests, or an error code if it's overloaded and doesn't have more capacity to serve requests.

The health endpoint is periodically queried by the load balancer. If the endpoint returns an error, the load balancer considers the process unhealthy and takes it out of the pool. Similarly, if the request to the health endpoint times out, the process is also taken out of the pool.

Health checks are critical to achieving high availability; if you have a service with 10 servers and one is unresponsive for some reason, then 10% of the requests will fail, which will cause the service's availability to drop to 90%.

Let's have a look at the different types of health checks that you can leverage in your service.

17.5.1 Health checks

A *liveness health test* is the most basic form of checking the health of a process. The load balancer simply performs a basic HTTP request to see whether the process replies with a *200 (OK)* status code.

A *local health test* checks whether the process is degraded or in some faulty state. The process's performance typically degrades when a local resource, like memory, CPU, or disk, is either close enough to be fully saturated, or is completely saturated. To detect a degradation, the process compares one or more local metrics, like memory available or remaining disk space, with some fixed upper and lower-bound thresholds. When a metric is above an upper-bound threshold, or below a lower-bound one, the process reports itself as unhealthy.

A more advanced, and also harder check to get right, is the *dependency health check*. This type of health check detects a degradation caused by a remote dependency, like a database, that needs to be accessed to handle incoming requests. The process measures the response time, timeouts, and errors of the remote calls directed to the dependency. If any measure breaks a predefined threshold, the process reports itself as unhealthy to reduce the load on the downstream dependency.

But here be dragons: if the downstream dependency is temporarily unreachable, or the health-check has a bug, then it's possible that all the processes behind the load balancer fail the health check. In that case, a naive load balancer would just take all service instances out of rotation, bringing the entire service down!

A smart load balancer instead detects that a large fraction of the service instances is being reported as unhealthy and considers the health check to no longer be reliable. Rather than continuing to remove processes from the pool, it starts to ignore the health-checks altogether so that new requests can be sent to any process in the pool.

17.6 Watchdog

One of the main reasons to build distributed services is to be able to withstand single-process failures. Since you are designing your system under the assumption that any process can crash at any time, your service needs to be able to deal with that eventuality.

For a process's crash to not affect your service's health, you should ensure ideally that:

- there are other processes that are identical to the one that crashed that can handle incoming requests;
- requests are stateless and can be served by any process;
- any non-volatile state is stored on a separate and dedicated data store so that when the process crashes its state isn't lost;
- all shared resources are leased so that when the process crashes, the leases expire and the resources can be accessed by other processes;

- the service is always running slightly over-scaled to withstand the occasional individual process failures.

Because crashes are inevitable and your service is prepared for them, you don't have to come up with complex recovery logic when a process gets into some weird degraded state — you can just let it crash. A transient but rare failure can be hard to diagnose and fix. Crashing and restarting the affected process gives operators maintaining the service some breathing room until the root-cause can be identified, giving the system a kind of self-healing property.

Imagine that a latent memory leak causes the available memory to decrease over time. When a process doesn't have more physical memory available, it starts to swap back and forth to the page file on disk. This swapping is extremely expensive and degrades the process's performance dramatically. If left unchecked, the memory leak would eventually bring all processes running the service on their knees. Would you rather have the processes detect they are degraded and restart themselves, or try to debug the root cause for the degradation at 3 AM?

To implement this pattern, a process should have a separate background thread that wakes up periodically — a watchdog — that monitors its health. For example, the watchdog could monitor the available physical memory left. When any monitored metric breaches a configured threshold, the watchdog considers the process degraded and deliberately restarts it.

The watchdog's implementation needs to be well-tested and monitored since a bug could cause the processes to restart continuously.

(PART) Testing and operations

Introduction

When all resiliency mechanisms fail, humans operators are the last line of defense. Historically, developers, testers, and operators were part of different teams. The developers handed over their software to a team of QA engineers responsible for testing it. When the software passed that stage, it moved to an operations team responsible for deploying it to production, monitoring it, and responding to alerts.

This model is being phased out in the industry as it has become commonplace for the development team to also be responsible for testing and operating the software they write. This forces the developers to embrace an end-to-end view of their applications, acknowledging that faults are inevitable and need to be accounted for.

Chapter 18 describes the different types of tests — unit, integration, and end-to-end tests — you can leverage to increase the confidence that your distributed applications work as expected.

Chapter 19 dives into continuous delivery and deployment pipelines used to release changes safely and efficiently to production.

Chapter 20 discusses how to use metrics and service-level indicators to monitor the health of distributed systems. It then describes how to define objectives that trigger alerts when breached. Finally, the chapter lists best practices for dashboard design.

Chapter 21 introduces the concept of observability and how it relates to monitoring. Then it describes how traces and logs can help developers debug their systems.

18 Testing

The longer it takes to detect a bug, the more expensive it becomes to fix it. Testing is all about catching bugs as early as possible, allowing developers to change the implementation with confidence that existing functionality won't break, increasing the speed of refactorings, shipping new features, and other changes. As a welcome side effect, testing also improves the system's design since developers have to put themselves in the users' shoes to test it effectively. Tests also provide up-to-date documentation.

Unfortunately, because it's impossible to predict all the ways a complex distributed application can fail, testing 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 only in production[1].

Although tests can't give you complete confidence that your code is bug-free, they certainly do a good job at detecting failure scenarios you are aware of and validating expected behaviors. As a rule of thumb, if you want to be confident that your implementation behaves in a certain way, you have to add a test for it.

18.1 Scope

Tests come in different shapes and sizes. To begin with, we need to distinguish between 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, and depending on it, the test can be categorized as either 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. A good unit test should be relatively static in time and change only when the behavior of the SUT changes — refactoring, fixing a

bug, or adding a new feature shouldn't require a unit test to change. To achieve that, a unit test should:

- use only the public interfaces of the SUT;
- test for state changes in the SUT (not predetermined sequence 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 its external dependencies as expected. This definition is not universal, though, because integration testing has different meanings for different people.

Martin Fowler makes the distinction between narrow and broad integration tests. A narrow integration test exercises only the code paths of a service that communicate with an external dependency, like the adapters and their 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. 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. Because of their scope, they are slow and more prone to intermittent failures.

End-to-end tests should not have any impact on other tests or users sharing the same environment. Among other things, that requires services to have good fault isolation mechanisms, like rate-limiting, to prevent buggy tests from affecting the rest of the system.

End-to-end tests can be painful and expensive to maintain. For example, when an end-to-end test fails, it's not always obvious which service is responsible and deeper investigation is required. But they 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 e-commerce service: create an order, modify it, and finally cancel it). Such a test usually requires less time to run than splitting the test into N 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 having any confidence in them and eventually ignore their failures. When possible, prefer 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 18.1).

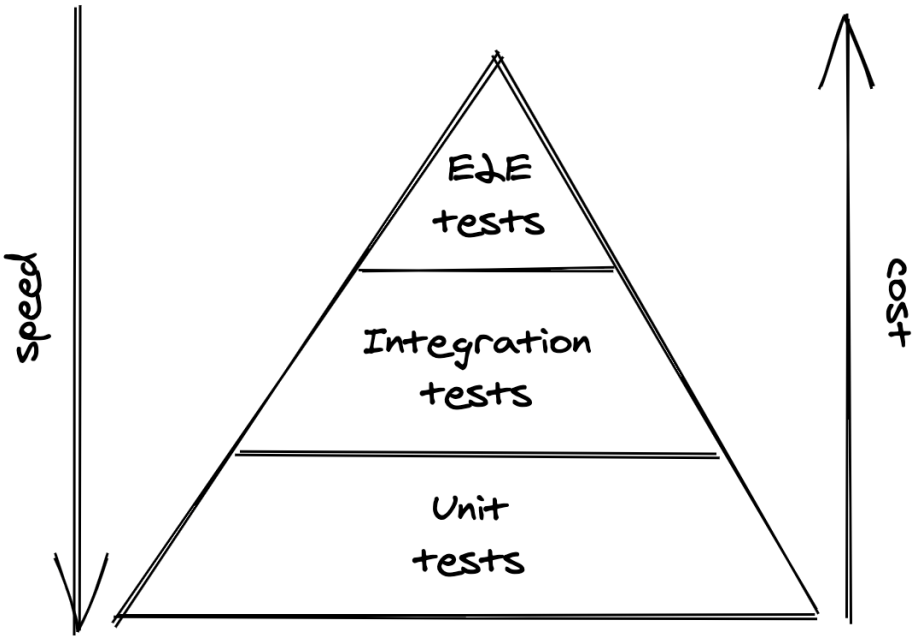


Figure 18.1: Test pyramid

18.2 Size

The size of a test 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 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. 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 you should write the smallest possible test for a given behavior. But how do you reduce the size of a test, while not reducing its scope?

You can use a *test double* in place of a real dependency to reduce the test's size, making it faster and less prone to intermittent failures. There are different types of test doubles:

- 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 less the resemblance is, the less confidence you should have that the test using the double is actually useful. Therefore, when the real implementation is fast, deterministic, and has few dependencies, use that rather than a double. If that's not the case,

you have to decide how realistic you want the test double to be, as there is a tradeoff between its fidelity and the test's size.

When using the real implementation is not an option, 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 use mocking with contract tests. A *contract test* defines the request it intends to send to an external dependency and the response it expects to receive from it. This contract is then used by the test to mock the external dependency. For example, a contract for a REST API consists of an HTTP request and response pair. To ensure that the contract doesn't break, the test suite of the external dependency uses the same contract to simulate a client and ensure that the expected response is returned.

18.3 Practical considerations

As with everything else, testing requires making tradeoffs.

Suppose we want to test the behavior of a specific user-facing API endpoint offered 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 18.2). As mentioned earlier, the general guideline is to write the smallest test possible with the desired scope.

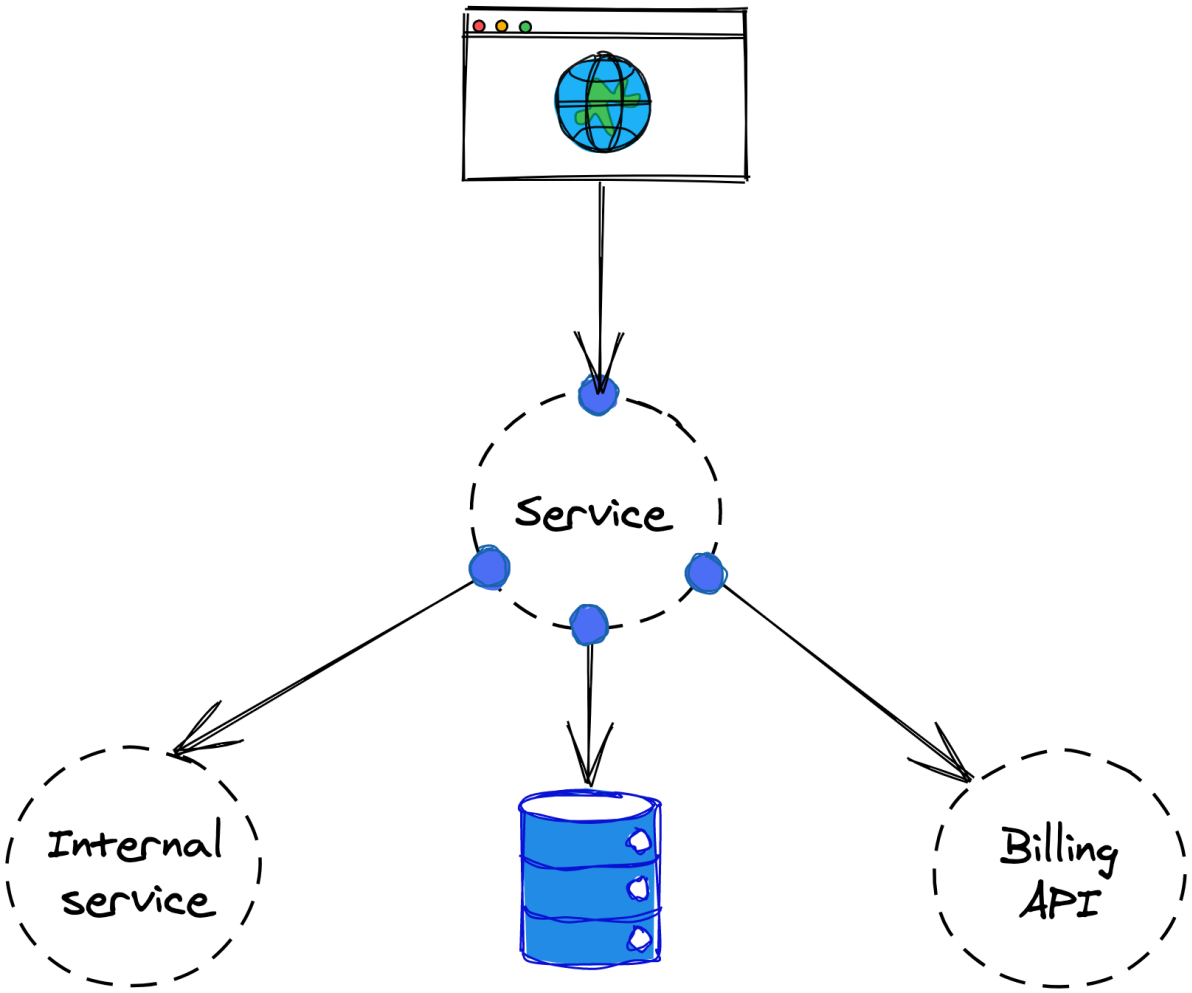


Figure 18.2: How would you test the service?

As it turns out, the endpoint doesn't need to communicate with the internal service, so we can safely use a mock in its place. The data store comes with an in-memory implementation (a fake) that we can leverage to avoid issuing network calls to a remote data store.

Finally, we can't use the third-party billing API, as that would require the test to issue real transactions. Fortunately, the API has a different endpoint that offers a playground environment, which the test can use without creating real transactions. If there was no playground environment available and no fake either, we would have to resort to stubbing or mocking.

In this case, we have cut the test's size considerably, while keeping its scope mostly intact.

Here is a more nuanced example. 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 for the functionality silently breaking is too high, we want to be as confident as possible that the functionality is working as expected. This warrants the use of an end-to-end test that runs in production and uses live services rather than test doubles.

---

1. Cindy Sridharan wrote a great blog post series on the topic at https://copyconstruct.medium.com/testing-microservices-the-sane-way-9bb31d158c16↩

19 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. Meaning that several changes, possibly over days or even weeks, end up being batched and released together. This makes it harder to pinpoint the breaking change[1] when a deployment fails, creating interruptions for the whole team. The developer who initiated the release also 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 delivery and deployment (CD) pipeline.

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.

19.1 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.

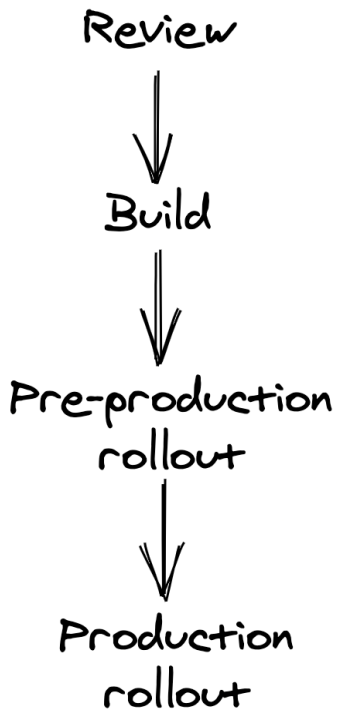


Figure 19.1: Continuous delivery and deployment pipeline stages

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, like e.g., unit tests, with larger tests only 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:

- 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, cloud resource templates, static assets, end-to-end tests, and configuration files should all be version-controlled in a repository (not necessarily the same) and be treated just like code. The same service can then have multiple CD pipelines, one for each repository, that can potentially run in parallel.

I can't stress enough the importance of reviewing and releasing configuration changes with a CD pipeline. One of the most common causes of production failures are configuration changes applied globally without any prior review or testing.

Once the change has been merged into the 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.

19.2 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.

You can even have 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, uses multiple pre-production environments (Alpha, Beta, and Gamma).

A service released to a pre-production environment should call the production endpoints of its external dependencies to make the environment as stable as possible; it could call the pre-production endpoints of other services owned by the same team, though.

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 to become a second-class citizen with sub-par health coverage.

19.3 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[2]. 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 the remaining instances need to pick up the slack. To avoid this causing 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 start with a low-traffic region first to reduce the impact of a faulty release. 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 will take 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.

19.4 Rollbacks

After each step, the CD pipeline needs to assess whether the artifact deployed is healthy, or else 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;
- alerts;
- and health endpoints.

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 can appear only after some time has passed. For example, a performance degradation could be visible only at peak time.

The CD pipeline can further gate the bake time on the number of requests seen for specific API endpoints to guarantee that the API surface has been properly exercised. To speed up the release, the bake time can be reduced after each step succeeds and confidence is built up.

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[3]. Based on their 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 a backward-incompatible change has been introduced.

Since rolling forward is much riskier than rolling back, any change introduced should always be backward compatible as a rule of thumb. The most common cause for backward-incompatibility is changing the serialization format used either for persistence or IPC purposes.

To safely introduce a backward-incompatible change, it needs to be broken down into multiple backward-compatible changes. 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 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 should only be 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 or not.

---

1. There could be multiple breaking changes actually.↩

2. This is also referred to as canary testing.↩

3. CD pipelines can be configured to run only during business hours to minimize the disruption to on-call engineers.↩

20 Monitoring

Monitoring is primarily used to detect failures that impact users in production and trigger notifications (alerts) sent to human operators responsible for mitigating them. The other critical use case for monitoring is to provide a high-level overview of the system's health through dashboards.

In the early days, monitoring was used mostly as a black-box approach to report whether a service was up or down, without much visibility of what was going on inside. Over the years, it has evolved into a white-box approach as developers started to instrument their code to emit application-level measurements to answer whether specific features worked as expected. This was popularized with the introduction of statsd by Etsy, which normalized collecting application-level measurements.

Blackbox monitoring is still in use today to monitor external dependencies, such as third-party APIs, and validate how the users perceive the performance and health of a service from the outside. A common approach is to periodically run scripts that send test requests to external API endpoints and monitor how long they took and whether they were successful. These scripts 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. These scripts are also useful to detect issues with APIs that aren't exercised often by users.

Blackbox monitoring is good at detecting the symptoms when something is broken; in contrast, white-box monitoring can help identify the root cause of known hard-failure modes before users are impacted. As a rule of thumb, if you can't design away a hard-failure mode, you should add monitoring for it. The longer a system has been around, the better you will understand how it can fail and what needs to be monitored.

20.1 Metrics

A *metric* is a numeric representation of information measured over a time interval and represented as a time-series, like the number of requests handled by a service. Conceptually, a metric is a list of samples, where each sample is represented by a floating-point number and a timestamp.

Modern monitoring systems allow a metric to be tagged with a set of key-value pairs called *labels*, which increases the dimensionality of the metric. Essentially, every distinct combination of labels is a different metric. This has become a necessity as modern services can have a large amount of metadata associated with each metric, like datacenter, cluster, node, pod, service, etc. High-cardinality metrics make it easy to slice and dice the data, and eliminate the instrumentation cost of manually creating a metric for each label combination.

A service should emit metrics about its load, internal state, and availability and performance of downstream service dependencies. Combined with the metrics emitted by downstream services, this allows operators to identify problems quickly. This requires explicit code changes and a deliberate 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 you will want to be able to answer once it's running in production[1]:

```
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 timeout?
  # 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 our service failed to handle. One way to do that is with an event-based approach — whenever a service instance fails to handle a request, it reports a failure count of 1 in an *event*[2] to a local telemetry agent, e.g.:

```
{
  "failureCount": 1,
  "serviceRegion": "EastUs2",
  "timestamp": 1614438079
}
```

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.

As you can imagine, this is quite expensive since the load on the backend increases with the number of events ingested. Events are also costly to aggregate at query time — suppose you want to retrieve the number of failures in North Europe over the past month; you would have to issue a query that requires fetching, filtering, and aggregating potentially trillions of events within that time period.

Is there a way to reduce costs at query time? Because metrics are time-series, they can be modeled and manipulated with mathematical tools. The samples of a time-series can be pre-aggregated over pre-specified time periods (e.g., 1 second, 5 minutes, 1 hour, etc.) and represented with summary statistics such as the sum, average, or percentiles.

For example, the telemetry backend can pre-aggregate metrics over one or more time periods at ingestion time. Conceptually, 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 backend can create multiple pre-aggregates with different periods. Then at query time, the pre-aggregated metric with the best period that satisfies the query is chosen. For example, CloudWatch (the telemetry backend 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 on the client side.

Client and server-side pre-aggregation drastically reduces bandwidth, compute, and storage requirements for metrics. However, it comes at a cost; operators lose the flexibility to re-aggregate metrics after they have been ingested, as they no longer have access to the original events that generated them. For example, if a metric is pre-aggregated over a period of time of 1 hour, it can't later be re-aggregated over a period of 5 min without the original events.

Because metrics are mainly used for alerting and visualization purposes, they are usually persisted in pre-aggregated form in a time-series data store since querying pre-aggregated data can be several order of magnitudes more efficient than the alternative.

20.2 Service-level indicators

As mentioned earlier, one of the main use cases for metrics is alerting. That doesn't mean we should create alerts for every possible metric out there — 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 for alerting.

A *service-level indicator* (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 average or percentile.

SLIs are best defined with a ratio of two metrics, good events over total number of events, since they are easy to interpret: 0 means the service is broken and 1 that everything is working as expected (see Figure 20.1). As we will see later in the chapter, ratios also simplify the configuration of alerts.

These are some commonly used SLIs for services:

- *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.
- *Quality* — The proportion of requests served in an un-degraded state (assuming the system degrades gracefully).
- *Data completeness* (for storage systems) — The proportion of records persisted to a data store that can be successfully accessed later.

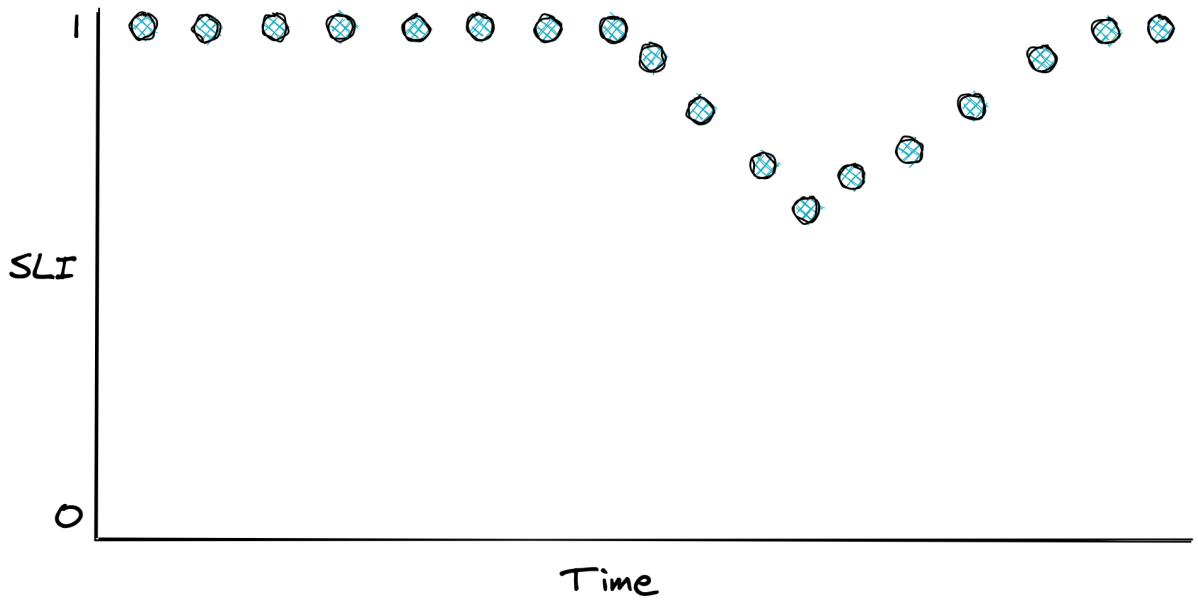


Figure 20.1: An SLI defined as the ratio of good events over the total number of events.

Once you have decided what to measure, you need to decide where to measure it. Take the response time, for example. Should you use the metric reported by the service, load balancer, or clients? In general, you want to use the one that best represents the experience of the users. And if that's too costly to collect, pick the next best candidate. In the previous example, the client metric is the more meaningful one, as that accounts for delays in the entire path of the request.

Now, how should you measure response times? Measurements can be affected by various factors that increase their variance, such as network timeouts, page faults, or heavy context switching. Since every request does not take the same amount of time, response times are best represented with a distribution, which tend to be right-skewed and long-tailed.

A distribution can be summarized with a statistic. Take the average, for example. While it has its uses, it doesn't tell you much about the proportion of requests experiencing a specific response time. All it takes is one extreme outlier to skew the average. For example, if 100 requests are hitting your service, 99 of which have a response time of 1 second and one of 10 min, the average is nearly 7 seconds. 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. In general, the higher the variance of a distribution is, the more likely the average user will be affected by long-tail behavior[3].

Even though only a small fraction of requests experience these extreme latencies, it impacts your most profitable users. They are the ones that make

the highest number of requests and thus have a higher chance of experiencing tail latencies. Several studies have shown that high latencies can negatively affect revenues. A mere 100-millisecond delay in load time can hurt conversion rates by 7 percent.

Also, long-tail behaviors left unchecked can quickly bring a service to its knees. Suppose a service is using 2K threads to serve 10K requests per second. By Little's Law, the average response time of a thread is 200 ms. Suddenly, a network switch becomes congested, and as it happens, 1% of requests are being served from a node behind that switch. That 1% of requests, or 100 requests per second out of the 10K, starts taking 20 seconds to complete.

How many more threads does the service need to deal with the small fraction of requests having a high response time? If 100 requests per second take 20 seconds to process, then 2K additional threads are needed to deal just with the slow requests. So the number of threads used by the service needs to double to keep up with the load!

Measuring long-tail behavior and keeping it under check doesn't just make your users happy, but also drastically improves the resiliency of your service and reduces operational costs. When you are forced to guard against the worst-case long-tail behavior, you happen to improve the average case as well.

20.3 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 20.2). An SLO sets the expectation to its users of how the service 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, typically 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.

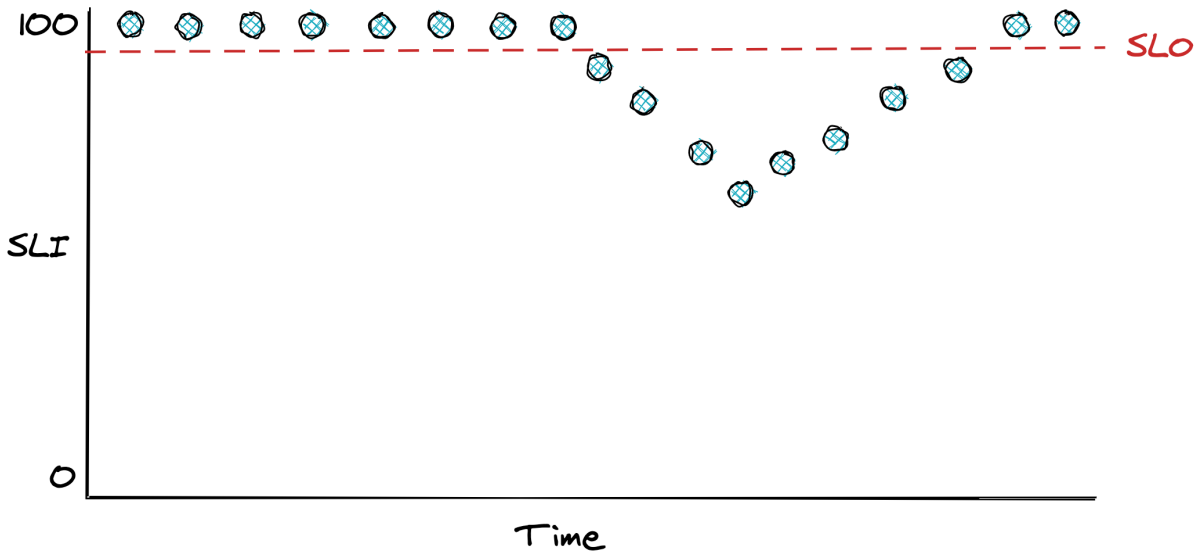


Figure 20.2: An SLO defines the range of acceptable values for an SLI.

SLOs are helpful for alerting purposes and help the team prioritize repair tasks with feature work. For example, the team can agree that when an error budget has been exhausted, repair items will take precedence over new features until the SLO is restored. Also, an incident's importance can be measured by how much of the error budget has been burned. An incident that burned 20% of the error budget needs more afterthought 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 in. Therefore 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 loose, you won't detect user-facing issues; if it's too strict, you will waste engineering time micro-optimizing and get diminishing returns. Even if you could guarantee 100% reliability for your system, you

can't make guarantees for anything that your users depend on to access your service that is outside your 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 your SLOs, start with comfortable ranges and tighten them as you build up confidence. Don't just pick targets that your service meets today that might become unattainable in a year after the load increases; work backward from what users care about. In general, anything above 3 nines of availability is very costly to achieve and provides diminishing returns.

How many SLOs should you have? You should strive to keep things simple and have as few as possible that provide a good enough indication of the desired service level. SLOs should also be documented and reviewed periodically. For example, suppose you discover that a specific user-facing issue generated lots of support tickets, but none of your SLOs showed any degradations. In that case, they are either too relaxed, or you are not measuring something that you should.

SLOs need to be agreed on with multiple stakeholders. Engineers need to agree that the targets are achievable without excessive toil. If the error budget is burning too rapidly or has been exhausted, repair items will take priority over features. Product managers have to agree that the targets guarantee a good user experience. As Google's SRE book mentions: "if you can't ever win a conversation about priorities by quoting a particular SLO, it's probably not worth having that SLO."

Users can become over-reliant on the actual behavior of your service rather than the published SLO. To mitigate that, you can consider injecting controlled failures in production — also known as chaos testing — to "shake the tree" and ensure the dependencies can cope with the targeted service level and are not making unrealistic assumptions. As an added benefit, injecting faults helps validate that resiliency mechanisms work as expected.

20.4 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 triggered 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 be mostly focusing on the latter case.

For an alert to be useful, it has to be actionable. The operator shouldn't spend time digging into 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 on the system without further investigation. On the other hand, an SLO is a good candidate for an alert because it quantifies its 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 we can discuss 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 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, you have to make a trade-off since improving one typically lowers the other.

Suppose you have an availability SLO of 99% over 30 days, and you 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 $\frac{1 \text{ hour}}{30 \text{ days}} = 0.14$. Is it really critical to be notified that 0.14% of the SLO's error budget has been burned? Probably not. In this case, you have high recall, but low precision.

You can improve the alert's precision by increasing the amount of time its condition needs to be true. The problem with it is that now the alert will take longer to trigger, which will be an issue when there is 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 increase of the error budget. Concretely, for our SLO 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.

By rearranging the burn rate's equation, you can derive the alert threshold that triggers when a specific percentage of the error budget has been burned. For example, to have an alert trigger when an error budget of 2% has been burned in a one-hour window, the threshold for the burn rate should be set to 14.4:

$$\begin{aligned} \text{error budget consumed} &= 0.02 \\ \text{time period elapsed} &= \frac{\text{alert window}}{\text{SLO period}} = \frac{1h}{720h} \\ \text{burn rate} &= \frac{\text{error budget consumed}}{\text{time period elapsed}} = 14.4 \end{aligned}$$

To improve recall, you can have multiple alerts with different thresholds. For example, a burn rate below 2 could be a low-severity alert that sends an e-mail and is investigated during working hours. The SRE workbook has some great examples of how to configure alerts based on burn rates.

While you should define most of your alerts based on SLOs, some should trigger for known hard-failure modes that you haven't had the time to design or debug away. For example, suppose you know your service suffers from a memory leak that has led to an incident in the past, but you haven't managed yet to track down the root-cause or build a resiliency mechanism to mitigate it. In this case, it could be useful to define an alert that triggers an automated restart when a service instance is running out of memory.

20.5 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, I will present some best practices on how to create useful dashboards.

The first decision you have to make when creating a dashboard is to decide who the audience is and what they are looking for. Given the audience, you can work backward to decide which charts, and therefore metrics, to include.

The categories of dashboards presented here (see Figure 20.3) are by no means standard but should give you an idea of how to organize dashboards.

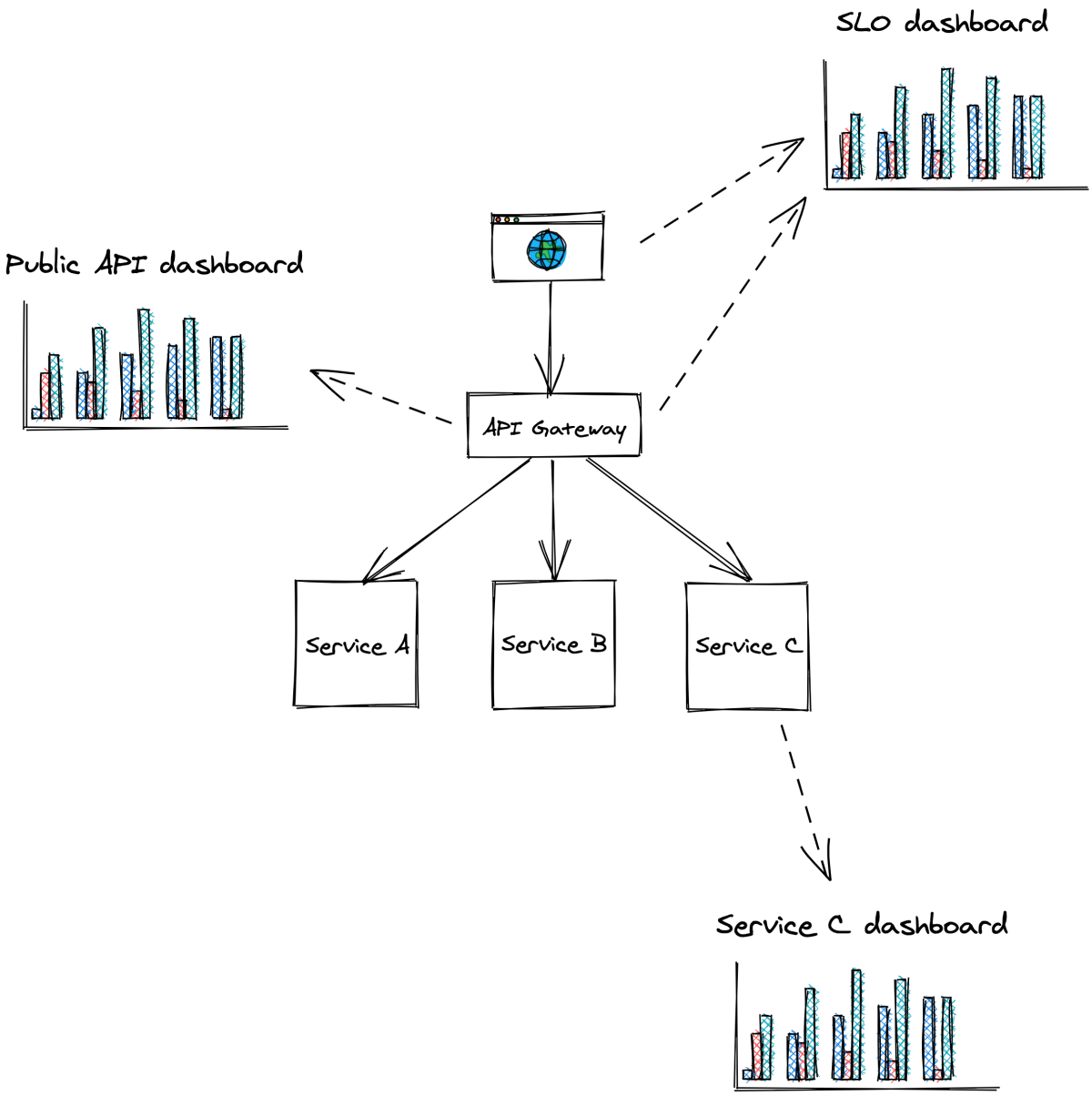


Figure 20.3: Dashboards should be tailored to their audience.

SLO dashboard

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 it's having on users.

Public API dashboard

This dashboard displays metrics about the system's public API endpoints, which helps operators identifying 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 messaged pulled from a messaging broker, size of requests, 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 a deep 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 for upstream dependencies like load balancers and messaging queues, and downstream dependencies like data stores.

This dashboard offers a first entry point into the behavior of a 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.

20.5.1 Best practices

As new metrics are added and old ones removed, charts and dashboards need to be modified and be kept in-sync across multiple environments like staging and production. The most effective way to achieve that is by defining dashboards and charts with a domain-specific language and version-control them just like code. This allows updating dashboards from the same pull request that contains related code changes without needing to update dashboards manually, which is error-prone.

As dashboards render top to bottom, the most important charts should always be located at the very top.

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.

Similarly, all charts in the same dashboard should use the same time resolution (e.g., 1 min, 5 min, 1 hour, etc.) and range (24 hours, 7 days, etc.). This makes it easy to correlate anomalies across charts in the same dashboard visually. You should pick the default time range and resolution based on the most common use case for a dashboard. For example, a 1-hour range with a 1-min resolution is best to monitor an ongoing incident, while a 1-year range with a 1-day resolution is best for capacity planning.

You should keep the number of data points and metrics on the same chart to a minimum. Rendering too many points doesn't just slow downloading charts, but also makes them hard to interpret 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, while 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, emit a metric using a value of zero in the absence of an error and a value of 1 in the presence of it.

20.6 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 same freedom usually enjoyed outside of regular working hours can cause 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 out of it and be given free rein to improve the on-call experience, for example, by revising dashboards or improving resiliency mechanisms.

Achieving a healthy on-call 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, as it's all too easy to miss a step when you get a call in the middle of the night[4]. Unless the alert was a false positive, all actions taken by the operator should be communicated into a shared channel like a global chat, that's accessible by other teams. This allows others to chime in, track the incident's progress, and make it easier to 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 brainstorm ways to prevent it from happening again. The more widespread the impact was, the more time you should spend on this. Incidents that burned a significant fraction of an SLO's error budget require a *postmortem*.

A postmortem's goal is to understand an incident's root cause and come up with a set of repair items that will prevent it from happening again. 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 provide a wealth of information and best practices regarding setting up a healthy on-call rotation.

---

1. I have omitted error handling for simplicity↩

2. We will talk more about event logs in section 21.1, for now assume an event is just a dictionary.↩

3. This tends to be primarily caused by various queues in the request-response path.↩

4. For the same reason, you should automate what you can to minimize manual actions that operators need to perform. Machines are good at following instructions; use that to your advantage.↩

21 Observability

A distributed system is never 100% healthy at any given time as there can always be 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. Unfortunately, they also increase the system's complexity. And with more complexity, it becomes increasingly harder to reason about the multitude of emergent behaviours the system might experience.

As we have discussed, human operators are still a fundamental part of operating a service as there are things that can't be automated, like mitigating an incident. Debugging is another example of such a task. When a system is designed to tolerate some level of degradation and self-heal, it's not necessary or possible to monitor every way it can get into an unhealthy state. You still need tooling and instrumentation to debug complex emergent failures because they are impossible to predict up-front.

When debugging, the operator makes an hypothesis and tries to validate it. For example, the operator might get suspicious after noticing that the variance of her 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 limit or a resource contention. 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 us to understand its emergent behaviours. A good observability platform strives to minimize the time it takes to validate hypotheses. This requires granular events with rich contexts, since it's not possible to know up-front what's going to 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 to deal with metrics that have many dimensions. Conversely, event logs and traces end up in transactional stores that can handle high-dimensional data well, but struggle with high throughput. Metrics are mainly used for monitoring, while event logs and traces mainly for debugging.

Observability is a superset of monitoring. While monitoring is focused exclusively on tracking the health of a system, observability also provides tools to understand and debug it. Monitoring on its own is good at detecting failure symptoms, but less so to explain their root cause (see Figure 21.1).

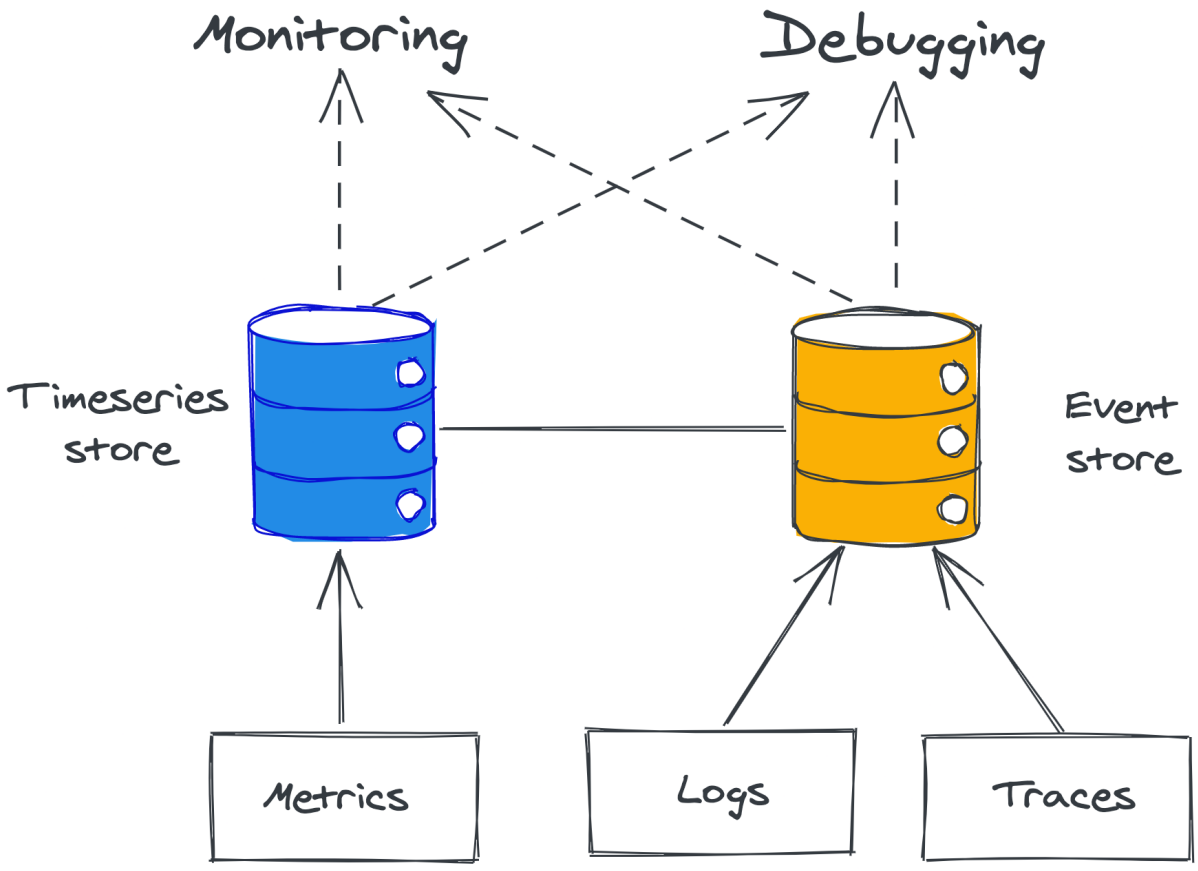


Figure 21.1: Observability is a superset of monitoring.

21.1 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 your services and external dependencies, like message brokers, proxies, databases, etc. Most languages offer libraries that make it easy to emit structured logs. Logs are typically dumped to disk files, which are rotated every so often, and forwarded by an agent to an external log collector asynchronously, like an ELK stack or AWS CloudWatch logs.

Logs provide a wealth of information about everything that's happening in a service. 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 to investigate long-tail behaviors that are missed by metrics represented with averages and percentiles, which can't help explain why a specific user request is failing.

Logs are very simple to emit, particularly free-form textual ones. But that's pretty much the only advantage they have compared to metrics and other instrumentation tools. Logging libraries can add overhead to your services if misused, especially when they are not asynchronous and logging blocks while writing to stdout or disk. Also, if the disk fills up due to excessive logging, the service instance might get itself into a degraded state. At best, you lose logging, and at worst, the service instance stops working if it requires disk access to handle requests.

Ingesting, processing, and storing a massive trove of data is not cheap either, no matter whether you 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 not less important, logs have a high noise to signal ratio because they are fine-grained and service-specific, which makes it challenging to extract useful information from them.

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 include measurements as well, like how long specific operations took. Every network call performed within the work unit needs to be instrumented to log its response status code and response time. Finally, data logged to the event should be sanitized and stripped of potentially sensitive properties that developers shouldn't have access to, like user content.

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, you 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 id of the request or message for the work unit.

Costs

There are various ways to keep the costs of logging under control. 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 is another option to reduce verbosity. For example, a service could log only one every n-th event. Additionally, events can also be prioritized based on their expected signal to noise ratio; for example, 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 you scale out and add more nodes, 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 killing your logging pipeline entirely, log collectors need to be able to rate-limit requests. If you use a third-party service to ingest, store, and query your logs, there probably is a quota in place already.

Of course, you can always opt to create in-memory aggregates from the measurements collected in events (e.g., metrics) and emit just those rather than raw logs. By doing so, you trade-off the ability to drill down into the aggregates if needed.

21.2 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* represent an interval of time that maps to a logical operation or work unit, and contains a bag of key-value pairs (see Figure 21.2).

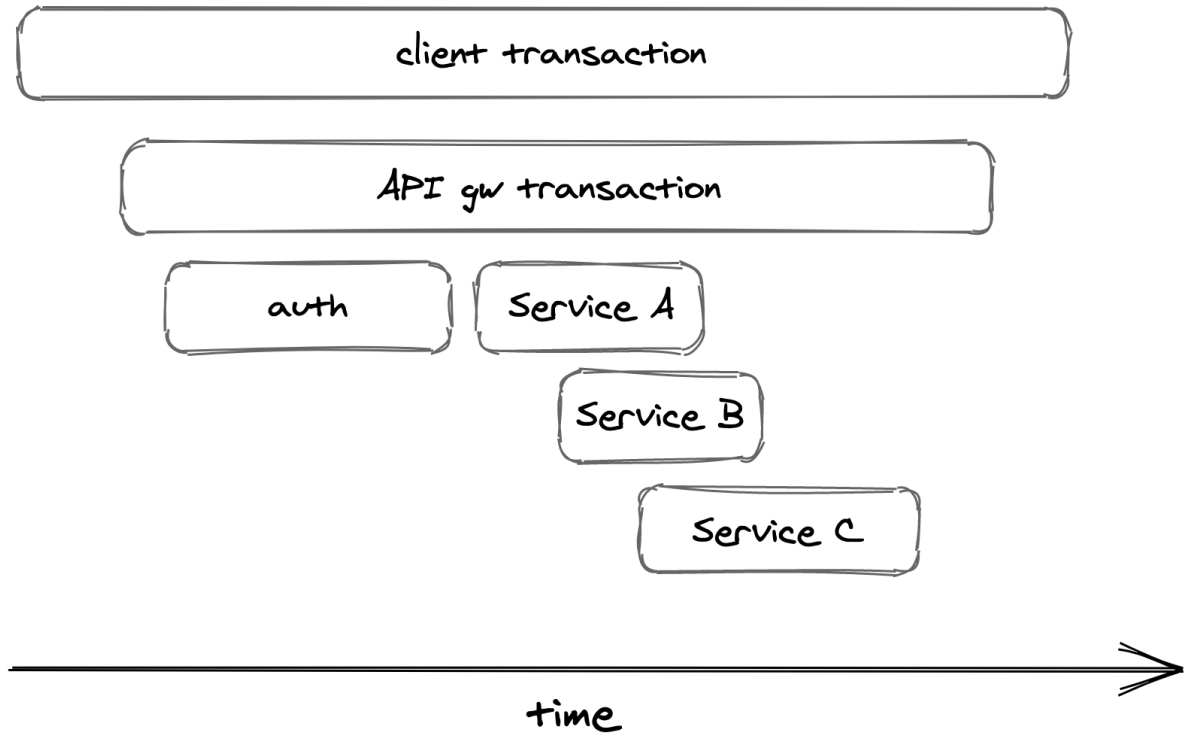


Figure 21.2: An execution flow can be represented with spans.

Traces allow developers to:

- debug issues affecting very specific requests, which can be used to investigate support requests;
- debug rare issues that affect only an extremely small fraction of requests;
- debug issues that affect a large fraction of requests, 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 clients hit which downstream services and in what proportion (also referred to as resource attribution), which can be used for rate-limiting or billing purposes.

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). 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 and AWS X-ray.

Tracing is challenging to retrofit into an existing system as it requires every component in the request path to be modified and propagate the trace context from one stage to the other. And it's not just the components that are under your control that need to support tracing; the frameworks and open source libraries you use need to support it as well, just like third-party services[1].

21.3 Putting it all together

The main drawback of event logs is that they are fine-grained and service-specific.

When a single 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 to debug the entire request flow. Similarly, a single event doesn't tell much about the health or state of a specific service.

This is where metrics and traces come in. You can think of them as abstractions, or derived views, built from event logs and tuned to specific use cases. A metric is a time-series of summary statistics derived by aggregating counters or observations over multiple work units or events. You could emit counters in events and have the backend roll them up into metrics as they are ingested. In fact, this is how some metrics 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, you can emit individual span events and have the backend aggregate them together into traces.

---

1. The service mesh pattern can help retrofit tracing.↩

22 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. One of the best ways to learn how to design large scale systems is by standing on the shoulders of the giants.

Industry papers provide a wealth of knowledge about distributed systems that have stood the test of time. My recommendation is to start with "Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency," which describes Azure's cloud storage system[1]. Azure's cloud storage is the core building block on top of which Microsoft built many other successful products. You will see many of the concepts introduced in the book there. One of the key design decisions was to guarantee strong consistency, unlike AWS S3[2], making the application developers' job much easier.

Once you have digested that, I suggest reading "Azure Data Explorer: a big data analytics cloud platform optimized for interactive, adhoc queries over structured, semi-structured and unstructured data." The 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[3].

Finally, if you are preparing for the system design interview, check out Alex Xu's book "System Design Interview." The book introduces a framework to tackle design interviews and includes more than 10 case studies.

---

1. Think of it as Azure's equivalent of AWS S3, which unfortunately doesn't have a public paper↩

2. S3 supports strong consistency since December 2020, though↩

3. I worked on a time-series data store that builds on top of Azure Data Explorer and Azure Storage, unfortunately, no public paper is available for it just yet↩