Roberto Vitillo

UNDERSTANDING DISTRIBUTED SYSTEMS

THE PRACTITIONER'S GUIDE TO
LARGE SCALE DISTRIBUTED APPLICATIONS

ROBERTO VITILLO

UNDERSTANDING DISTRIBUTED SYSTEMS

VERSION 1.0.2

Contents

Copyright 11

About the author 13

Acknowledgements 15

Preface 17

0.1 Who should read this book 17

1 Introduction 19

1.1 Communication 20

1.2 Coordination 20

1.3 Scalability 21

1.4 Resiliency 21

1.5 Operations 22

1.6 Anatomy of a distributed system 23

I Communication 27

2 Reliable links 31

2.1 Reliability 31

2.2 Connection lifecycle 31

2.3 Flow control 33

2.4 Congestion control 34

2.5 Custom protocols 35

3 Secure links 37
3.1 Encryption 37
3.2 Authentication 38
3.3 Integrity 39
3.4 Handshake 40

4 Discovery 41

5 APIs 45
5.1 HTTP 46
5.2 Resources 48
5.3 Request methods 50
5.4 Response status codes 51
5.5 OpenAPI 52
5.6 Evolution 53

II Coordination 55

6 System models 59

7 Failure detection 63

8 Time 65
8.1 Physical clocks 65
8.2 Logical clocks 66
8.3 Vector clocks 68

9 Leader election 71
9.1 Raft leader election 71
9.2 Practical considerations 72

10 Replication 77
10.1 State machine replication 77
10.2 Consensus 80

10.3 Consistency models 80
10.3.1 Strong consistency 82
10.3.2 Sequential consistency 83
10.3.3 Eventual consistency 84
10.3.4 CAP theorem 84
10.4 Practical considerations 85

11 Transactions 87
11.1 ACID 87
11.2 Isolation 88
11.2.1 Concurrency control 90
11.3 Atomicity 91
11.3.1 Two-phase commit 91
11.4 Asynchronous transactions 93
11.4.1 Log-based transactions 93
11.4.2 Sagas 96
11.4.3 Isolation 97

III Scalability 99

12 Functional decomposition 103
12.1 Microservices 103
12.1.1 Benefits 105
12.1.2 Costs 105
12.1.3 Practical considerations 107
12.2 API gateway 108
12.2.1 Routing 109
12.2.2 Composition 109
12.2.3 Translation 109
12.2.4 Cross-cutting concerns 110
12.2.5 Caveats 112
12.3 CQRS 113
12.4 Messaging 115
12.4.1 Guarantees 117
12.4.2 Exactly-once processing 118
12.4.3 Failures 119
12.4.4 Backlogs 119
12.4.5 Fault isolation 120

12.4.6 Reference plus blob 120

13 Partitioning 123
13.1 Sharding strategies 123
13.1.1 Range partitioning 123
13.1.2 Hash partitioning 124
13.2 Rebalancing 127
13.2.1 Static partitioning 127
13.2.2 Dynamic partitioning 127
13.2.3 Practical considerations 127

14 Duplication 129
14.1 Network load balancing 129
14.1.1 DNS load balancing 131
14.1.2 Transport layer load balancing 132
14.1.3 Application layer load balancing 133
14.1.4 Geo load balancing 135
14.2 Replication 137
14.2.1 Single leader replication 137
14.2.2 Multi-leader replication 139
14.2.3 Leaderless replication 141
14.3 Caching 142
14.3.1 Policies 142
14.3.2 In-process cache 143
14.3.3 Out-of-process cache 144

IV Resiliency 147

15 Common failure causes 151
15.1 Single point of failure 151
15.2 Unreliable network 152
15.3 Slow processes 152
15.4 Unexpected load 153
15.5 Cascading failures 154
15.6 Risk management 155

16 Downstream resiliency 157
16.1 Timeout 157
16.2 Retry 159
16.2.1 Exponential backoff 160
16.2.2 Retry amplification 161
16.3 Circuit breaker 162
16.3.1 State machine 162

17 Upstream resiliency 165
17.1 Load shedding 165
17.2 Load leveling 166
17.3 Rate-rimiting 167
17.3.1 Single-process implementation 168
17.3.2 Distributed implementation 170
17.4 Bulkhead 171
17.5 Health endpoint 173
17.5.1 Health checks 174
17.6 Watchdog 174

V Testing and operations 177

18 Testing 181
18.1 Scope 181
18.2 Size 183
18.3 Practical considerations 184

19 Continuous delivery and deployment 187
19.1 Review and build 188
19.2 Pre-production 188
19.3 Production 189
19.4 Rollbacks 190

20 Monitoring 193
20.1 Metrics 194
20.2 Service-level indicators 196
20.3 Service-level objectives 198

20.4 Alerts 200
20.5 Dashboards 202
20.5.1 Best practices 204
20.6 On-call 205

21 Observability 207
21.1 Logs 208
21.2 Traces 210
21.3 Putting it all together 212

22 Final words 213

Copyright

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[1], the best-paid engineering roles require distributed systems expertise. That comes as no surprise as modern applications are distributed systems.

[1] https://insights.stackoverflow.com/survey/2020#work-salary-by-developer-type-united-states

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[2] 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.

[2] I plan to update the book regularly, which is why it has a version number. No book is ever perfect, and 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 at roberto@understandingdistributed.systems

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

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[1], and you need to understand how the stack works when that happens.

[1] https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/

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 mes-

sengers 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.

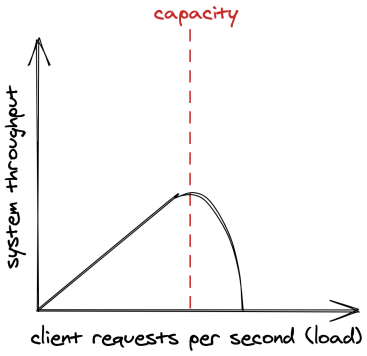

Figure 1.1: The system throughput on the y axis is the subset of client requests (x axis) that can be handled without errors and with low response times, also referred to as its goodput.

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*[2] 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

[2] `http://wiki.c2.com/?PortsAndAdaptersArchitecture`

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.

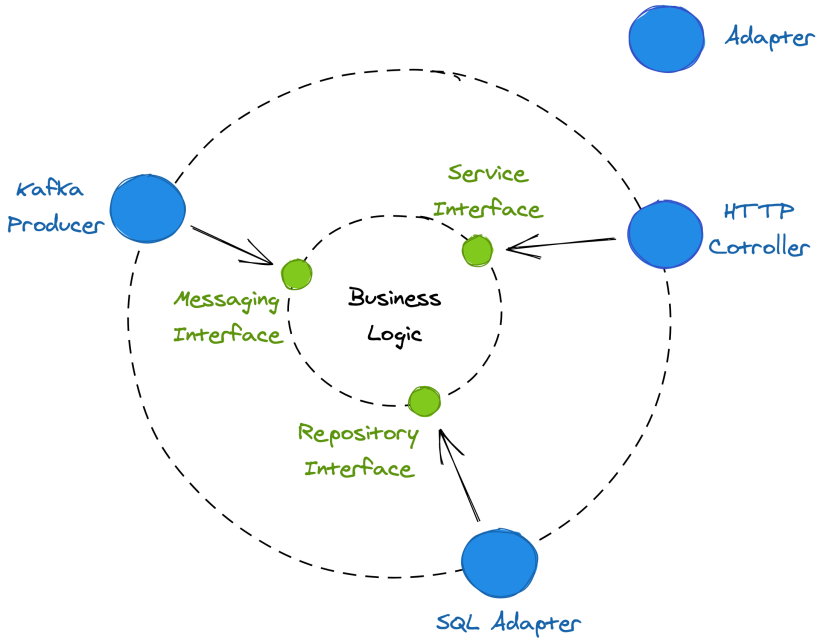

Figure 1.2: The business logic uses the messaging interface implemented by the Kafka producer to send messages and the repository interface to access the SQL store. In contrast, the HTTP controller handles incoming requests using the service interface.

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.

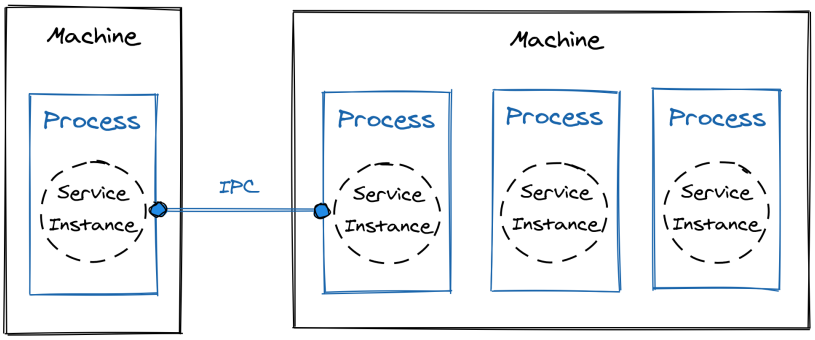

Figure 1.3: The different architectural points of view used in this book.

Part I

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[3], 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.

[3] `https://en.wikipedia.org/wiki/Internet_protocol_suite`

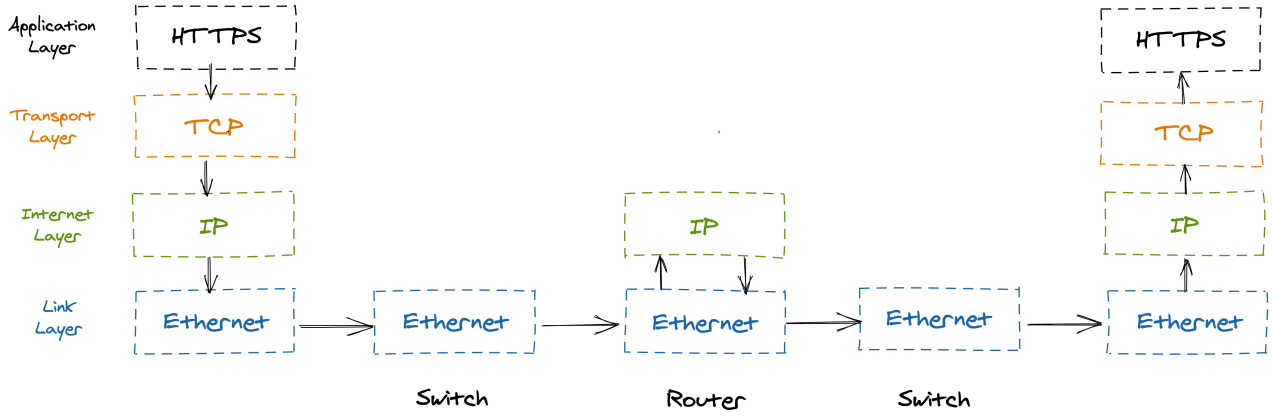

Figure 1.4: Internet protocol suite

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[1] 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.

[1] https://tools.ietf.org/html/rfc793

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[2] than the

[2] https://en.wikipedia.org/wiki/Transmission_Control_Protocol#/media/File:Tcp_state_diagram_fixed_new.svg

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.

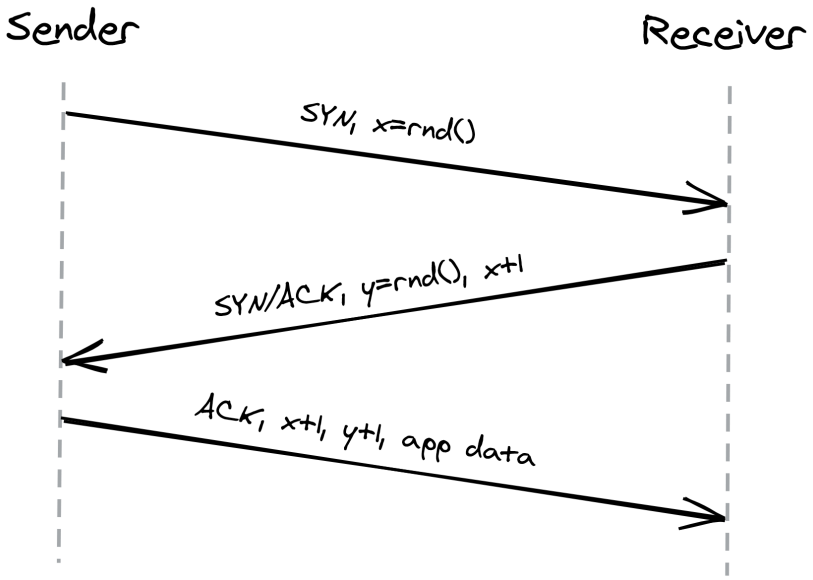

Figure 2.1: Three-way handshake

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:

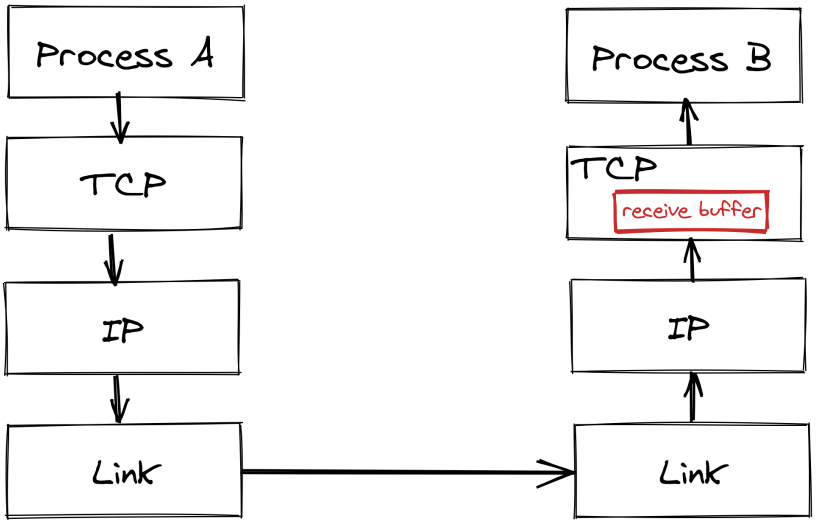

Figure 2.2: The receive buffer stores data that hasn't been processed yet by the application.

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.

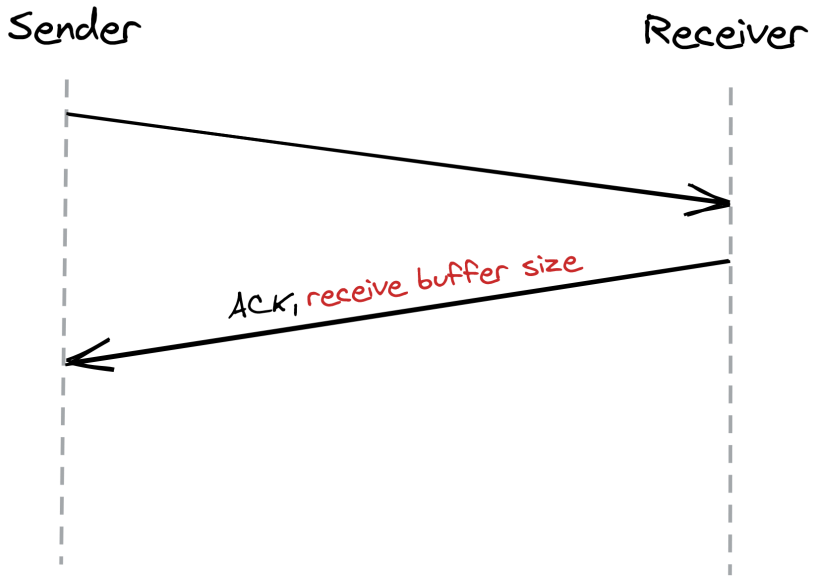

Figure 2.3: The size of the receive buffer is communicated in the headers of acknowledgments segments.

This mechanism is not too dissimilar to rate-limiting[3] at the service level. But, rather than rate-limiting on an API key or IP address, TCP is rate-limiting on a connection level.

[3] https://en.wikipedia.org/wiki/Rate_limiting

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.

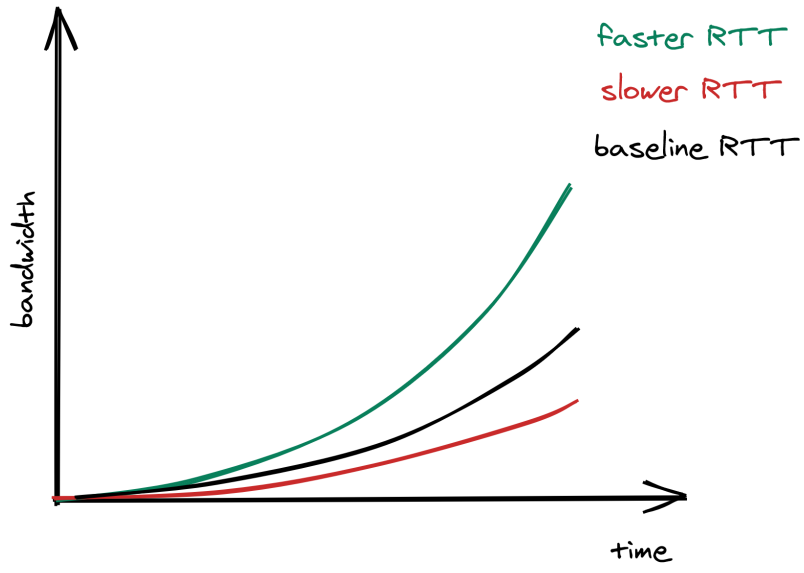

Figure 2.4: The lower the RTT is, the quicker the sender can start utilizing the underlying network's bandwidth.

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[4] by a certain amount, and timeouts decrease it by another.

[4] https://en.wikipedia.org/wiki/CUBIC_TCP

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[5] shows that bandwidth is a function of latency. TCP will try very hard to optimize the window size since it can't do anything about the round trip time. However, that doesn't always yield the optimal configuration. Due to the way congestion control works, the 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.

[5] https://en.m.wikipedia.org/wiki/Bandwidth-delay_product

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*[6] (UDP) — a connectionless transport layer protocol that can be used as an alternative to TCP.

[6] https://en.wikipedia.org/wiki/User_Datagram_Protocol

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[7].

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

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.

GET THE REST OF THE BOOK AT

HTTPS://UNDERSTANDINGDISTRIBUTED.SYSTEMS/