Chapter 9. The Trouble with Distributed Systems

> *They're funny things, Accidents. You never have them till you're having them.*
>
> —A.A. Milne, *The House at Pooh Corner* (1928)

A NOTE FOR EARLY RELEASE READERS

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 9th chapter of the final book. The GitHub repo for this book is *https://github.com/ept/ddia2-feedback*.

If you'd like to be actively involved in reviewing and commenting on this draft, please reach out on GitHub.

As discussed in "Reliability and Fault Tolerance", making a system reliable means ensuring that the system as a whole continues working, even when things go wrong (i.e., when there is a fault). However, anticipating all the possible faults and handling them is not that easy. As a developer, it is very tempting to focus mostly on the happy path (after all, most of the time things work fine!) and to neglect faults, since they introduce a lot of edge cases.

If you want your system to be reliable in the presence of faults you have to radically change your mindset, and focus on the things that could go wrong, even though they may be unlikely. It doesn't matter whether there is only a one-in-a-million chance of a thing going wrong: in a large enough system, one-in-a-million events happen every day. Experienced systems operators will tell you that anything that *can* go wrong *will* go wrong.

Moreover, working with distributed systems is fundamentally different from writing software on a single computer—and the main difference is that there are lots of new and exciting ways for things to go wrong [1, 2]. In this chapter, you will get a taste of the problems that arise in practice, and an understanding of the things you can and cannot rely on.

To understand what challenges we are up against, we will now turn our pessimism to the maximum and explore the things that may go wrong in a distributed system. We will look into problems with networks ("Unreliable Networks") as well as clocks and timing issues ("Unreliable Clocks"). The consequences of all these issues are disorienting, so we'll explore how to think about the state of a distributed system and how to reason about things that have happened ("Knowledge, Truth, and Lies"). Later, in Chapter 10, we will look at some examples of how we can achieve fault tolerance in the face of those faults.

Faults and Partial Failures

When you are writing a program on a single computer, it normally behaves in a fairly predictable way: either it works or it doesn't. Buggy software may give the appearance that the computer is sometimes "having a bad day" (a problem that is often fixed by a reboot), but that is mostly just a consequence of badly written software.

There is no fundamental reason why software on a single computer should be flaky: when the hardware is working correctly, the same operation always produces the same result (it is *deterministic*). If there is a hardware problem (e.g., memory corruption or a loose connector), the consequence is usually a total system failure (e.g., kernel panic, "blue screen of death," failure to start up). An individual computer with good software is usually either fully functional or entirely broken, but not something in between.

This is a deliberate choice in the design of computers: if an internal fault occurs, we prefer a computer to crash completely rather than returning a wrong result, because wrong results are difficult and confusing to deal with. Thus, computers hide the fuzzy physical reality on which they are implemented and present an idealized system model that operates with mathematical perfection. A CPU instruction always does the same thing; if you write some data to memory or disk, that data remains intact and doesn't

get randomly corrupted. As discussed in "Hardware and Software Faults", this is not actually true—in reality, data does get silently corrupted and CPUs do sometimes silently return the wrong result—but it happens rarely enough that we can get away with ignoring it.

When you are writing software that runs on several computers, connected by a network, the situation is fundamentally different. In distributed systems, faults occur much more frequently, and so we can no longer ignore them—we have no choice but to confront the messy reality of the physical world. And in the physical world, a remarkably wide range of things can go wrong, as illustrated by this anecdote [3]:

> *In my limited experience I've dealt with long-lived network partitions in a single data center (DC), PDU [power distribution unit] failures, switch failures, accidental power cycles of whole racks, whole-DC backbone failures, whole-DC power failures, and a hypoglycemic driver smashing his Ford pickup truck into a DC's HVAC [heating, ventilation, and air conditioning] system. And I'm not even an ops guy.*
>
> —Coda Hale

In a distributed system, there may well be some parts of the system that are broken in some unpredictable way, even though other parts of the system are working fine. This is known as a *partial failure*. The difficulty is that partial failures are *nondeterministic*: if you try to do anything involving multiple nodes and the network, it may sometimes work and sometimes unpredictably fail. As we shall see, you may not even *know* whether something succeeded or not!

This nondeterminism and possibility of partial failures is what makes distributed systems hard to work with [4]. On the other hand, if a distributed system can tolerate partial failures, that opens up powerful possibilities: for example, it allows you to perform a rolling upgrade, rebooting one node at a time to install software updates while the system as a whole continues working uninterrupted all the time. Fault tolerance therefore allows us to make distributed systems more reliable than single-node systems: we can build a reliable system from unreliable components.

But before we can implement fault tolerance, we need to know more about the faults that we're supposed to tolerate. It is important to consider a wide range

of possible faults—even fairly unlikely ones—and to artificially create such situations in your testing environment to see what happens. In distributed systems, suspicion, pessimism, and paranoia pay off.

Unreliable Networks

In the past, older computers such as mainframes were made reliable by ensuring individual components were redundant, for example by employing RAID to sustain individual disk failures. As discussed in "Shared-Memory, Shared-Disk, and Shared-Nothing Architecture", the distributed systems we focus on in this book are mostly *shared-nothing systems*: i.e., a bunch of machines connected by a network. Instead of having redundancy of components within a single machine, shared-nothing systems use replication across separate machines for redundancy. The network is the only way these machines can communicate—we assume that each machine has its own memory and disk, and one machine cannot access another machine's memory or disk (except by making requests to a service over the network). Even when storage is shared, such as with object storage, machines communicate with shared storage services over the network.

The internet and most internal networks in datacenters (often Ethernet) are *asynchronous packet networks*. In this kind of network, one node can send a message (a packet) to another node, but the network gives no guarantees as to when it will arrive, or whether it will arrive at all. If you send a request and expect a response, many things could go wrong (some of which are illustrated in Figure 9-1):

1. Your request may have been lost (perhaps someone unplugged a network cable).
2. Your request may be waiting in a queue and will be delivered later (perhaps the network or the recipient is overloaded).
3. The remote node may have failed (perhaps it crashed or it was powered down).
4. The remote node may have temporarily stopped responding (perhaps it is experiencing a long garbage collection pause; see "Process Pauses"), but it will start responding again later.
5. The remote node may have processed your request, but the response has been lost on the network (perhaps a network switch has been misconfigured).

6. The remote node may have processed your request, but the response has been delayed and will be delivered later (perhaps the network or your own machine is overloaded).

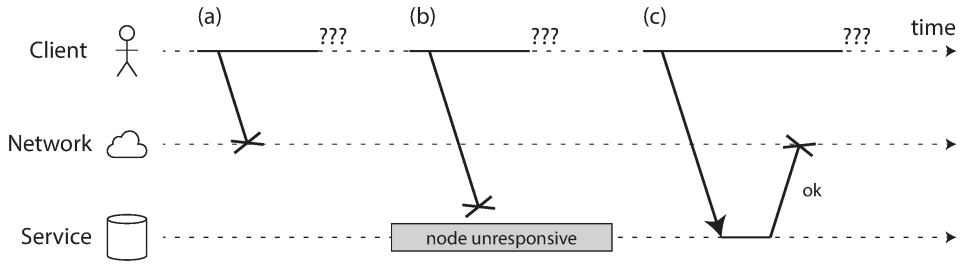

Figure 9-1. If you send a request and don't get a response, it's not possible to distinguish whether (a) the request was lost, (b) the remote node is down, or (c) the response was lost.

The sender can't even tell whether the packet was delivered: the only option is for the recipient to send a response message, which may in turn be lost or delayed. These issues are indistinguishable in an asynchronous network: the only information you have is that you haven't received a response yet. If you send a request to another node and don't receive a response, it is *impossible* to tell why.

The usual way of handling this issue is a *timeout*: after some time you give up waiting and assume that the response is not going to arrive. However, when a timeout occurs, you still don't know whether the remote node got your request or not (and if the request is still queued somewhere, it may still be delivered to the recipient, even if the sender has given up on it).

The Limitations of TCP

Network packets have a maximum size (generally a few kilobytes), but many applications need to send messages (requests, responses) that are too big to fit in one packet. These applications most often use TCP, the Transmission Control Protocol, to establish a *connection* that breaks up large data streams into individual packets, and puts them back together again on the receiving side.

NOTE

Most of what we say about TCP applies also to its more recent alternative QUIC, as well as the Stream Control Transmission Protocol (SCTP) used in WebRTC, the BitTorrent uTP protocol, and other transport protocols. For a comparison to UDP, see "TCP Versus UDP".

TCP is often described as providing "reliable" delivery, in the sense that it detects and retransmits dropped packets, it detects reordered packets and puts them back in the correct order, and it detects packet corruption using a simple checksum. It also figures out how fast it can send data so that it is transferred as quickly as possible, but without overloading the network or the receiving node; this is known as *congestion control*, *flow control*, or *backpressure* [5].

When you "send" some data by writing it to a socket, it actually doesn't get sent immediately, but it's only placed in a buffer managed by your operating system. When the congestion control algorithm decides that it has capacity to send a packet, it takes the next packet-worth of data from that buffer and passes it to the network interface. The packet passes through several switches and routers, and eventually the receiving node's operating system places the packet's data in a receive buffer and sends an acknowledgment packet back to the sender. Only then does the receiving operating system notify the application that some more data has arrived [6].

So, if TCP provides "reliability", does that mean we no longer need to worry about networks being unreliable? Unfortunately not. It decides that a packet must have been lost if no acknowledgment arrives within some timeout, but TCP can't tell either whether it was the outbound packet or the acknowledgment that was lost. Although TCP can resend the packet, it can't guarantee that the new packet will get through either. If the network cable is unplugged, TCP can't plug it back in for you. Eventually, after a configurable timeout, TCP gives up and signals an error to the application. TCP's deduplication and retransmission capabilities only apply to a single connection, so if the application reconnects and retransmits, data could be duplicated.

If a TCP connection is closed with an error—perhaps because the remote node crashed, or perhaps because the network was interrupted—you unfortunately have no way of knowing how much data was actually processed by the remote node [6]. Even if TCP acknowledged that a packet was delivered, this only

means that the operating system kernel on the remote node received it, but the application may have crashed before it handled that data. If you want to be sure that a request was successful, you need a positive response from the application itself [7].

Nevertheless, TCP is very useful, because it provides a convenient way of sending and receiving messages that are too big to fit in one packet. Once a TCP connection is established, you can also use it to send multiple requests and responses. This is usually done by first sending a header that indicates the length of the following message in bytes, followed by the actual message. HTTP and many RPC protocols (see "Dataflow Through Services: REST and RPC") work like this.

Network Faults in Practice

We have been building computer networks for decades—one might hope that by now we would have figured out how to make them reliable. Unfortunately, we have not yet succeeded. There are some systematic studies, and plenty of anecdotal evidence, showing that network problems can be surprisingly common, even in controlled environments like a datacenter operated by one company [8]:

- One study in a medium-sized datacenter found about 12 network faults per month, of which half disconnected a single machine, and half disconnected an entire rack [9].
- Another study measured the failure rates of components like top-of-rack switches, aggregation switches, and load balancers [10]. It found that adding redundant networking gear doesn't reduce faults as much as you might hope, since it doesn't guard against human error (e.g., misconfigured switches), which is a major cause of outages.
- Interruptions of wide-area fiber links have been blamed on cows [11], beavers [12], and sharks [13] (though shark bites have become rarer due to better shielding of submarine cables [14]). Humans are also at fault, be it due to accidental misconfiguration [15], scavenging [16], or sabotage [17].
- Across different cloud regions, round-trip times of up to several *minutes* have been observed at high percentiles [18, Table 3]. Even within a single datacenter, packet delay of more than a minute can occur during a network topology reconfiguration, triggered by a problem during a software

upgrade for a switch [19]. Thus, we have to assume that messages might be delayed arbitrarily.

- Sometimes communications are partially interrupted, depending on who you're talking to: for example, A and B can communicate, B and C can communicate, but A and C cannot [20, 21]. Other surprising faults include a network interface that sometimes drops all inbound packets but sends outbound packets successfully [22]: just because a network link works in one direction doesn't guarantee it's also working in the opposite direction.
- Even a brief network interruption can have repercussions that last for much longer than the original issue [8, 20, 23].

NETWORK PARTITIONS

When one part of the network is cut off from the rest due to a network fault, that is sometimes called a *network partition* or *netsplit*, but it is not fundamentally different from other kinds of network interruption. Network partitions are not related to sharding of a storage system, which is sometimes also called *partitioning* (see Chapter 7).

Even if network faults are rare in your environment, the fact that faults *can* occur means that your software needs to be able to handle them. Whenever any communication happens over a network, it may fail—there is no way around it.

If the error handling of network faults is not defined and tested, arbitrarily bad things could happen: for example, the cluster could become deadlocked and permanently unable to serve requests, even when the network recovers [24], or it could even delete all of your data [25]. If software is put in an unanticipated situation, it may do arbitrary unexpected things.

Handling network faults doesn't necessarily mean *tolerating* them: if your network is normally fairly reliable, a valid approach may be to simply show an error message to users while your network is experiencing problems. However, you do need to know how your software reacts to network problems and ensure that the system can recover from them. It may make sense to deliberately trigger network problems and test the system's response (this is known as *fault injection*; see "Fault injection").

Detecting Faults

Many systems need to automatically detect faulty nodes. For example:

- A load balancer needs to stop sending requests to a node that is dead (i.e., take it *out of rotation*).
- In a distributed database with single-leader replication, if the leader fails, one of the followers needs to be promoted to be the new leader (see "Handling Node Outages").

Unfortunately, the uncertainty about the network makes it difficult to tell whether a node is working or not. In some specific circumstances you might get some feedback to explicitly tell you that something is not working:

- If you can reach the machine on which the node should be running, but no process is listening on the destination port (e.g., because the process crashed), the operating system will helpfully close or refuse TCP connections by sending a `RST` or `FIN` packet in reply.
- If a node process crashed (or was killed by an administrator) but the node's operating system is still running, a script can notify other nodes about the crash so that another node can take over quickly without having to wait for a timeout to expire. For example, HBase does this [26].
- If you have access to the management interface of the network switches in your datacenter, you can query them to detect link failures at a hardware level (e.g., if the remote machine is powered down). This option is ruled out if you're connecting via the internet, or if you're in a shared datacenter with no access to the switches themselves, or if you can't reach the management interface due to a network problem.
- If a router is sure that the IP address you're trying to connect to is unreachable, it may reply to you with an ICMP Destination Unreachable packet. However, the router doesn't have a magic failure detection capability either—it is subject to the same limitations as other participants of the network.

Rapid feedback about a remote node being down is useful, but you can't count on it. If something has gone wrong, you may get an error response at some level of the stack, but in general you have to assume that you will get no response at all. You can retry a few times, wait for a timeout to elapse, and eventually declare the node dead if you don't hear back within the timeout. Since the node could actually be alive, we must balance between false

positives and false negatives: too short a timeout causes alive nodes to be incorrectly suspected as dead, and too long a timeout causes unnecessary delays waiting for dead nodes.

Timeouts and Unbounded Delays

If a timeout is the only sure way of detecting a fault, then how long should the timeout be? There is unfortunately no simple answer.

A long timeout means a long wait until a node is declared dead (and during this time, users may have to wait or see error messages). A short timeout detects faults faster, but carries a higher risk of incorrectly declaring a node dead when in fact it has only suffered a temporary slowdown (e.g., due to a load spike on the node or the network).

Prematurely declaring a node dead is problematic: if the node is actually alive and in the middle of performing some action (for example, sending an email), and another node takes over, the action may end up being performed twice. We will discuss this issue in more detail in "Knowledge, Truth, and Lies", and in Chapters 10 and 12.

When a node is declared dead, its responsibilities need to be transferred to other nodes, which places additional load on other nodes and the network. If the system is already struggling with high load, declaring nodes dead prematurely can make the problem worse. In particular, it could happen that the node actually wasn't dead but only slow to respond due to overload; transferring its load to other nodes can cause a cascading failure (in the extreme case, all nodes declare each other dead, and everything stops working—see "When an Overloaded System Won't Recover").

Imagine a fictitious system with a network that guaranteed a maximum delay for packets—every packet is either delivered within some time d, or it is lost, but delivery never takes longer than d. Furthermore, assume that you can guarantee that a non-failed node always handles a request within some time r. In this case, you could guarantee that every successful request receives a response within time $2d + r$—and if you don't receive a response within that time, you know that either the network or the remote node is not working. If this was true, $2d + r$ would be a reasonable timeout to use.

Unfortunately, most systems we work with have neither of those guarantees: asynchronous networks have *unbounded delays* (that is, they try to deliver packets as quickly as possible, but there is no upper limit on the time it may take for a packet to arrive), and most server implementations cannot guarantee that they can handle requests within some maximum time (see "Response time guarantees"). For failure detection, it's not sufficient for the system to be fast most of the time: if your timeout is low, it only takes a transient spike in round-trip times to throw the system off-balance.

Network congestion and queueing

When driving a car, travel times on road networks often vary most due to traffic congestion. Similarly, the variability of packet delays on computer networks is most often due to queueing [27]:

- If several different nodes simultaneously try to send packets to the same destination, the network switch must queue them up and feed them into the destination network link one by one (as illustrated in Figure 9-2). On a busy network link, a packet may have to wait a while until it can get a slot (this is called *network congestion*). If there is so much incoming data that the switch queue fills up, the packet is dropped, so it needs to be resent—even though the network is functioning fine.
- When a packet reaches the destination machine, if all CPU cores or application threads are currently busy, the incoming request from the network is queued by the operating system until the application is ready to handle it. Depending on the load on the machine, this may take an arbitrary length of time [28].
- In virtualized environments, a running operating system is often paused for tens of milliseconds while another virtual machine uses a CPU core. During this time, the VM cannot consume any data from the network, so the incoming data is queued (buffered) by the virtual machine monitor [29], further increasing the variability of network delays.
- As mentioned earlier, in order to avoid overloading the network, TCP limits the rate at which it sends data. This means additional queueing at the sender before the data even enters the network.

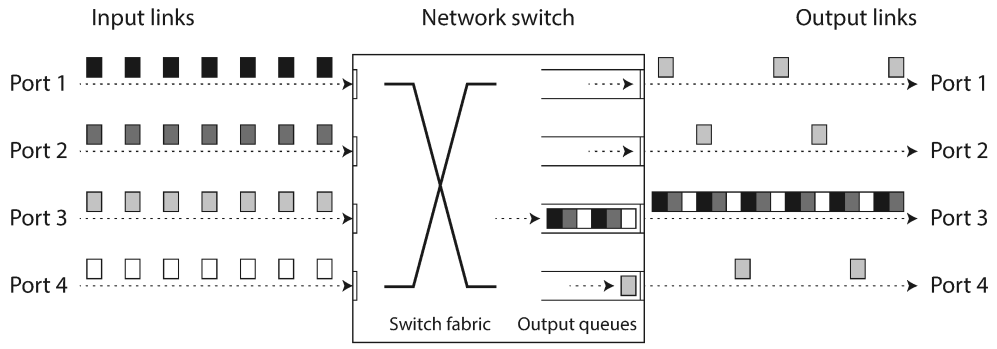

Figure 9-2. If several machines send network traffic to the same destination, its switch queue can fill up. Here, ports 1, 2, and 4 are all trying to send packets to port 3.

Moreover, when TCP detects and automatically retransmits a lost packet, although the application does not see the packet loss directly, it does see the resulting delay (waiting for the timeout to expire, and then waiting for the retransmitted packet to be acknowledged).

TCP VERSUS UDP

Some latency-sensitive applications, such as videoconferencing and Voice over IP (VoIP), use UDP rather than TCP. It's a trade-off between reliability and variability of delays: as UDP does not perform flow control and does not retransmit lost packets, it avoids some of the reasons for variable network delays (although it is still susceptible to switch queues and scheduling delays).

UDP is a good choice in situations where delayed data is worthless. For example, in a VoIP phone call, there probably isn't enough time to retransmit a lost packet before its data is due to be played over the loudspeakers. In this case, there's no point in retransmitting the packet—the application must instead fill the missing packet's time slot with silence (causing a brief interruption in the sound) and move on in the stream. The retry happens at the human layer instead. ("Could you repeat that please? The sound just cut out for a moment.")

All of these factors contribute to the variability of network delays. Queueing delays have an especially wide range when a system is close to its maximum capacity: a system with plenty of spare capacity can easily drain queues, whereas in a highly utilized system, long queues can build up very quickly.

In public clouds and multitenant datacenters, resources are shared among many customers: the network links and switches, and even each machine's

network interface and CPUs (when running on virtual machines), are shared. Processing large amounts of data can use the entire capacity of network links (*saturate* them). As you have no control over or insight into other customers' usage of the shared resources, network delays can be highly variable if someone near you (a *noisy neighbor*) is using a lot of resources [30, 31].

In such environments, you can only choose timeouts experimentally: measure the distribution of network round-trip times over an extended period, and over many machines, to determine the expected variability of delays. Then, taking into account your application's characteristics, you can determine an appropriate trade-off between failure detection delay and risk of premature timeouts.

Even better, rather than using configured constant timeouts, systems can continually measure response times and their variability (*jitter*), and automatically adjust timeouts according to the observed response time distribution. The Phi Accrual failure detector [32], which is used for example in Akka and Cassandra [33] is one way of doing this. TCP retransmission timeouts also work similarly [5].

Synchronous Versus Asynchronous Networks

Distributed systems would be a lot simpler if we could rely on the network to deliver packets with some fixed maximum delay, and not to drop packets. Why can't we solve this at the hardware level and make the network reliable so that the software doesn't need to worry about it?

To answer this question, it's interesting to compare datacenter networks to the traditional fixed-line telephone network (non-cellular, non-VoIP), which is extremely reliable: delayed audio frames and dropped calls are very rare. A phone call requires a constantly low end-to-end latency and enough bandwidth to transfer the audio samples of your voice. Wouldn't it be nice to have similar reliability and predictability in computer networks?

When you make a call over the telephone network, it establishes a *circuit*: a fixed, guaranteed amount of bandwidth is allocated for the call, along the entire route between the two callers. This circuit remains in place until the call ends [34]. For example, an ISDN network runs at a fixed rate of 4,000 frames per second. When a call is established, it is allocated 16 bits of space within each frame (in each direction). Thus, for the duration of the call, each side is

guaranteed to be able to send exactly 16 bits of audio data every 250 microseconds [35].

This kind of network is *synchronous*: even as data passes through several routers, it does not suffer from queueing, because the 16 bits of space for the call have already been reserved in the next hop of the network. And because there is no queueing, the maximum end-to-end latency of the network is fixed. We call this a *bounded delay*.

Can we not simply make network delays predictable?

Note that a circuit in a telephone network is very different from a TCP connection: a circuit is a fixed amount of reserved bandwidth which nobody else can use while the circuit is established, whereas the packets of a TCP connection opportunistically use whatever network bandwidth is available. You can give TCP a variable-sized block of data (e.g., an email or a web page), and it will try to transfer it in the shortest time possible. While a TCP connection is idle, it doesn't use any bandwidth (except perhaps for an occasional keepalive packet).

If datacenter networks and the internet were circuit-switched networks, it would be possible to establish a guaranteed maximum round-trip time when a circuit was set up. However, they are not: Ethernet and IP are packet-switched protocols, which suffer from queueing and thus unbounded delays in the network. These protocols do not have the concept of a circuit.

Why do datacenter networks and the internet use packet switching? The answer is that they are optimized for *bursty traffic*. A circuit is good for an audio or video call, which needs to transfer a fairly constant number of bits per second for the duration of the call. On the other hand, requesting a web page, sending an email, or transferring a file doesn't have any particular bandwidth requirement—we just want it to complete as quickly as possible.

If you wanted to transfer a file over a circuit, you would have to guess a bandwidth allocation. If you guess too low, the transfer is unnecessarily slow, leaving network capacity unused. If you guess too high, the circuit cannot be set up (because the network cannot allow a circuit to be created if its bandwidth allocation cannot be guaranteed). Thus, using circuits for bursty data transfers wastes network capacity and makes transfers unnecessarily

slow. By contrast, TCP dynamically adapts the rate of data transfer to the available network capacity.

There have been some attempts to build hybrid networks that support both circuit switching and packet switching. *Asynchronous Transfer Mode* (ATM) was a competitor to Ethernet in the 1980s, but it didn't gain much adoption outside of telephone network core switches. InfiniBand has some similarities [36]: it implements end-to-end flow control at the link layer, which reduces the need for queueing in the network, although it can still suffer from delays due to link congestion [37]. With careful use of *quality of service* (QoS, prioritization and scheduling of packets) and *admission control* (rate-limiting senders), it is possible to emulate circuit switching on packet networks, or provide statistically bounded delay [27, 34]. New network algorithms like Low Latency, Low Loss, and Scalable Throughput (L4S) attempt to mitigate some of the queuing and congestion control problems both at the client and router level. Linux's traffic controller (TC) also allows applications to reprioritize packets for QoS purposes.

More generally, you can think of variable delays as a consequence of dynamic resource partitioning.

Say you have a wire between two telephone switches that can carry up to 10,000 simultaneous calls. Each circuit that is switched over this wire occupies one of those call slots. Thus, you can think of the wire as a resource that can be shared by up to 10,000 simultaneous users. The resource is divided up in a *static* way: even if you're the only call on the wire right now, and all other 9,999 slots are unused, your circuit is still allocated the same fixed amount of bandwidth as when the wire is fully utilized.

By contrast, the internet shares network bandwidth *dynamically*. Senders push and jostle with each other to get their packets over the wire as quickly as possible, and the network switches decide which packet to send (i.e., the bandwidth allocation) from one moment to the next. This approach has the downside of queueing, but the advantage is that it maximizes utilization of the wire. The wire has a fixed cost, so if you utilize it better, each byte you send over the wire is cheaper.

A similar situation arises with CPUs: if you share each CPU core dynamically between several threads, one thread sometimes has to wait in the operating system's run queue while another thread is running, so a thread can be paused for varying lengths of time [38]. However, this utilizes the hardware better than if you allocated a static number of CPU cycles to each thread (see "Response time guarantees"). Better hardware utilization is also why cloud platforms run several virtual machines from different customers on the same physical machine.

Latency guarantees are achievable in certain environments, if resources are statically partitioned (e.g., dedicated hardware and exclusive bandwidth allocations). However, it comes at the cost of reduced utilization—in other words, it is more expensive. On the other hand, multitenancy with dynamic resource partitioning provides better utilization, so it is cheaper, but it has the downside of variable delays.

Variable delays in networks are not a law of nature, but simply the result of a cost/benefit trade-off.

However, such quality of service is currently not enabled in multitenant datacenters and public clouds, or when communicating via the internet. Currently deployed technology does not allow us to make any guarantees about delays or reliability of the network: we have to assume that network congestion, queueing, and unbounded delays will happen. Consequently, there's no "correct" value for timeouts—they need to be determined experimentally.

Peering agreements between internet service providers and the establishment of routes through the Border Gateway Protocol (BGP), bear closer resemblance to circuit switching than IP itself. At this level, it is possible to buy dedicated bandwidth. However, internet routing operates at the level of networks, not individual connections between hosts, and at a much longer timescale.

Unreliable Clocks

Clocks and time are important. Applications depend on clocks in various ways to answer questions like the following:

1. Has this request timed out yet?
2. What's the 99th percentile response time of this service?
3. How many queries per second did this service handle on average in the last five minutes?
4. How long did the user spend on our site?
5. When was this article published?
6. At what date and time should the reminder email be sent?
7. When does this cache entry expire?
8. What is the timestamp on this error message in the log file?

Examples 1–4 measure *durations* (e.g., the time interval between a request being sent and a response being received), whereas examples 5–8 describe *points in time* (events that occur on a particular date, at a particular time).

In a distributed system, time is a tricky business, because communication is not instantaneous: it takes time for a message to travel across the network from one machine to another. The time when a message is received is always later than the time when it is sent, but due to variable delays in the network, we don't know how much later. This fact sometimes makes it difficult to

determine the order in which things happened when multiple machines are involved.

Moreover, each machine on the network has its own clock, which is an actual hardware device: usually a quartz crystal oscillator. These devices are not perfectly accurate, so each machine has its own notion of time, which may be slightly faster or slower than on other machines. It is possible to synchronize clocks to some degree: the most commonly used mechanism is the Network Time Protocol (NTP), which allows the computer clock to be adjusted according to the time reported by a group of servers [39]. The servers in turn get their time from a more accurate time source, such as a GPS receiver.

Monotonic Versus Time-of-Day Clocks

Modern computers have at least two different kinds of clocks: a *time-of-day clock* and a *monotonic clock*. Although they both measure time, it is important to distinguish the two, since they serve different purposes.

Time-of-day clocks

A time-of-day clock does what you intuitively expect of a clock: it returns the current date and time according to some calendar (also known as *wall-clock time*). For example, `clock_gettime(CLOCK_REALTIME)` on Linux and `System.currentTimeMillis()` in Java return the number of seconds (or milliseconds) since the *epoch*: midnight UTC on January 1, 1970, according to the Gregorian calendar, not counting leap seconds. Some systems use other dates as their reference point. (Although the Linux clock is called *real-time*, it has nothing to do with real-time operating systems, as discussed in "Response time guarantees".)

Time-of-day clocks are usually synchronized with NTP, which means that a timestamp from one machine (ideally) means the same as a timestamp on another machine. However, time-of-day clocks also have various oddities, as described in the next section. In particular, if the local clock is too far ahead of the NTP server, it may be forcibly reset and appear to jump back to a previous point in time. These jumps, as well as similar jumps caused by leap seconds, make time-of-day clocks unsuitable for measuring elapsed time [40].

Time-of-day clocks can experience jumps due to the start and end of Daylight Saving Time (DST); these can be avoided by always using UTC as time zone,

which does not have DST. Time-of-day clocks have also historically had quite a coarse-grained resolution, e.g., moving forward in steps of 10 ms on older Windows systems [41]. On recent systems, this is less of a problem.

Monotonic clocks

A monotonic clock is suitable for measuring a duration (time interval), such as a timeout or a service's response time: `clock_gettime(CLOCK_MONOTONIC)` or `clock_gettime(CLOCK_BOOTTIME)` on Linux [42] and `System.nanoTime()` in Java are monotonic clocks, for example. The name comes from the fact that they are guaranteed to always move forward (whereas a time-of-day clock may jump back in time).

You can check the value of the monotonic clock at one point in time, do something, and then check the clock again at a later time. The *difference* between the two values tells you how much time elapsed between the two checks — more like a stopwatch than a wall clock. However, the *absolute* value of the clock is meaningless: it might be the number of nanoseconds since the computer was booted up, or something similarly arbitrary. In particular, it makes no sense to compare monotonic clock values from two different computers, because they don't mean the same thing.

On a server with multiple CPU sockets, there may be a separate timer per CPU, which is not necessarily synchronized with other CPUs [43]. Operating systems compensate for any discrepancy and try to present a monotonic view of the clock to application threads, even as they are scheduled across different CPUs. However, it is wise to take this guarantee of monotonicity with a pinch of salt [44].

NTP may adjust the frequency at which the monotonic clock moves forward (this is known as *slewing* the clock) if it detects that the computer's local quartz is moving faster or slower than the NTP server. By default, NTP allows the clock rate to be speeded up or slowed down by up to 0.05%, but NTP cannot cause the monotonic clock to jump forward or backward. The resolution of monotonic clocks is usually quite good: on most systems they can measure time intervals in microseconds or less.

In a distributed system, using a monotonic clock for measuring elapsed time (e.g., timeouts) is usually fine, because it doesn't assume any synchronization

between different nodes' clocks and is not sensitive to slight inaccuracies of measurement.

Clock Synchronization and Accuracy

Monotonic clocks don't need synchronization, but time-of-day clocks need to be set according to an NTP server or other external time source in order to be useful. Unfortunately, our methods for getting a clock to tell the correct time aren't nearly as reliable or accurate as you might hope—hardware clocks and NTP can be fickle beasts. To give just a few examples:

- The quartz clock in a computer is not very accurate: it *drifts* (runs faster or slower than it should). Clock drift varies depending on the temperature of the machine. Google assumes a clock drift of up to 200 ppm (parts per million) for its servers [45], which is equivalent to 6 ms drift for a clock that is resynchronized with a server every 30 seconds, or 17 seconds drift for a clock that is resynchronized once a day. This drift limits the best possible accuracy you can achieve, even if everything is working correctly.
- If a computer's clock differs too much from an NTP server, it may refuse to synchronize, or the local clock will be forcibly reset [39]. Any applications observing the time before and after this reset may see time go backward or suddenly jump forward.
- If a node is accidentally firewalled off from NTP servers, the misconfiguration may go unnoticed for some time, during which the drift may add up to large discrepancies between different nodes' clocks. Anecdotal evidence suggests that this does happen in practice.
- NTP synchronization can only be as good as the network delay, so there is a limit to its accuracy when you're on a congested network with variable packet delays. One experiment showed that a minimum error of 35 ms is achievable when synchronizing over the internet [46], though occasional spikes in network delay lead to errors of around a second. Depending on the configuration, large network delays can cause the NTP client to give up entirely.
- Some NTP servers are wrong or misconfigured, reporting time that is off by hours [47, 48]. NTP clients mitigate such errors by querying several servers and ignoring outliers. Nevertheless, it's somewhat worrying to bet the correctness of your systems on the time that you were told by a stranger on the internet.

- Leap seconds result in a minute that is 59 seconds or 61 seconds long, which messes up timing assumptions in systems that are not designed with leap seconds in mind [49]. The fact that leap seconds have crashed many large systems [40, 50] shows how easy it is for incorrect assumptions about clocks to sneak into a system. The best way of handling leap seconds may be to make NTP servers "lie," by performing the leap second adjustment gradually over the course of a day (this is known as *smearing*) [51, 52], although actual NTP server behavior varies in practice [53]. Leap seconds will no longer be used from 2035 onwards, so this problem will fortunately go away.
- In virtual machines, the hardware clock is virtualized, which raises additional challenges for applications that need accurate timekeeping [54]. When a CPU core is shared between virtual machines, each VM is paused for tens of milliseconds while another VM is running. From an application's point of view, this pause manifests itself as the clock suddenly jumping forward [29]. If a VM pauses for several seconds, the clock may then be several seconds behind the actual time, but NTP may continue to report that the clock is almost perfectly in sync [55].
- If you run software on devices that you don't fully control (e.g., mobile or embedded devices), you probably cannot trust the device's hardware clock at all. Some users deliberately set their hardware clock to an incorrect date and time, for example to cheat in games [56]. As a result, the clock might be set to a time wildly in the past or the future.

It is possible to achieve very good clock accuracy if you care about it sufficiently to invest significant resources. For example, the MiFID II European regulation for financial institutions requires all high-frequency trading funds to synchronize their clocks to within 100 microseconds of UTC, in order to help debug market anomalies such as "flash crashes" and to help detect market manipulation [57].

Such accuracy can be achieved with some special hardware (GPS receivers and/or atomic clocks), the Precision Time Protocol (PTP) and careful deployment and monitoring [58, 59]. Relying on GPS alone can be risky because GPS signals can easily be jammed. In some locations this happens frequently, e.g. close to military facilities [60]. Some cloud providers have begun offering high-accuracy clock synchronization for their virtual machines [61]. However, clock synchronization still requires a lot of care. If your NTP daemon is misconfigured, or a firewall is blocking NTP traffic, the clock error due to drift can quickly become large.

Relying on Synchronized Clocks

The problem with clocks is that while they seem simple and easy to use, they have a surprising number of pitfalls: a day may not have exactly 86,400 seconds, time-of-day clocks may move backward in time, and the time according to one node's clock may be quite different from another node's clock.

Earlier in this chapter we discussed networks dropping and arbitrarily delaying packets. Even though networks are well behaved most of the time, software must be designed on the assumption that the network will occasionally be faulty, and the software must handle such faults gracefully. The same is true with clocks: although they work quite well most of the time, robust software needs to be prepared to deal with incorrect clocks.

Part of the problem is that incorrect clocks easily go unnoticed. If a machine's CPU is defective or its network is misconfigured, it most likely won't work at all, so it will quickly be noticed and fixed. On the other hand, if its quartz clock is defective or its NTP client is misconfigured, most things will seem to work fine, even though its clock gradually drifts further and further away from reality. If some piece of software is relying on an accurately synchronized clock, the result is more likely to be silent and subtle data loss than a dramatic crash [62, 63].

Thus, if you use software that requires synchronized clocks, it is essential that you also carefully monitor the clock offsets between all the machines. Any node whose clock drifts too far from the others should be declared dead and removed from the cluster. Such monitoring ensures that you notice the broken clocks before they can cause too much damage.

Timestamps for ordering events

Let's consider one particular situation in which it is tempting, but dangerous, to rely on clocks: ordering of events across multiple nodes [64]. For example, if two clients write to a distributed database, who got there first? Which write is the more recent one?

Figure 9-3 illustrates a dangerous use of time-of-day clocks in a database with multi-leader replication (the example is similar to Figure 6-8). Client A writes

$x = 1$ on node 1; the write is replicated to node 3; client B increments x on node 3 (we now have $x = 2$); and finally, both writes are replicated to node 2.

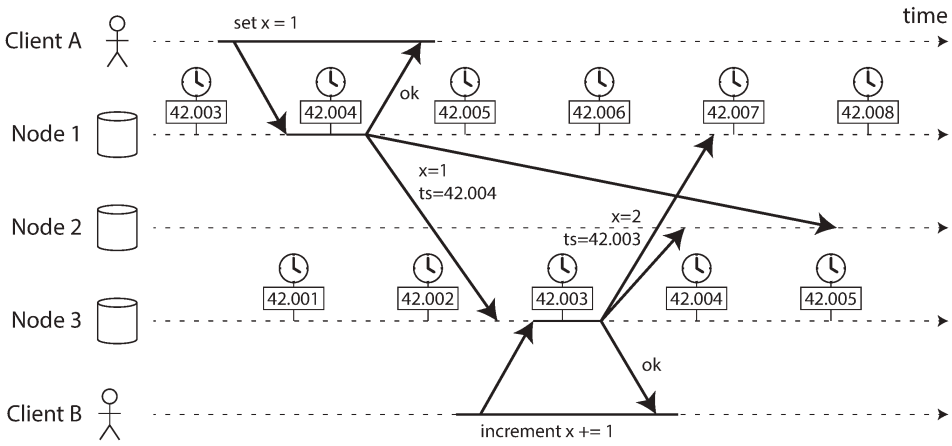

Figure 9-3. The write by client B is causally later than the write by client A, but B's write has an earlier timestamp.

In Figure 9-3, when a write is replicated to other nodes, it is tagged with a timestamp according to the time-of-day clock on the node where the write originated. The clock synchronization is very good in this example: the skew between node 1 and node 3 is less than 3 ms, which is probably better than you can expect in practice.

Since the increment builds upon the earlier write of $x = 1$, we might expect that the write of $x = 2$ should have the greater timestamp of the two. Unfortunately, that is not what happens in Figure 9-3: the write $x = 1$ has a timestamp of 42.004 seconds, but the write $x = 2$ has a timestamp of 42.003 seconds.

As discussed in "Last write wins (discarding concurrent writes)", one way of resolving conflicts between concurrently written values on different nodes is *last write wins* (LWW), which means keeping the write with the greatest timestamp for a given key and discarding all writes with older timestamps. In the example of Figure 9-3, when node 2 receives these two events, it will incorrectly conclude that $x = 1$ is the more recent value and drop the write $x = 2$, so the increment is lost.

This problem can be prevented by ensuring that when a value is overwritten, the new value always has a higher timestamp than the overwritten value, even if that timestamp is ahead of the writer's local clock. However, that incurs the cost of an additional read to find the greatest existing timestamp. Some systems, including Cassandra and ScyllaDB, want to write to all replicas in a single round trip, and therefore they simply use the client clock's timestamp

along with a last write wins policy [62]. This approach has some serious problems:

- Database writes can mysteriously disappear: a node with a lagging clock is unable to overwrite values previously written by a node with a fast clock until the clock skew between the nodes has elapsed [63, 65]. This scenario can cause arbitrary amounts of data to be silently dropped without any error being reported to the application.
- LWW cannot distinguish between writes that occurred sequentially in quick succession (in Figure 9-3, client B's increment definitely occurs *after* client A's write) and writes that were truly concurrent (neither writer was aware of the other). Additional causality tracking mechanisms, such as version vectors, are needed in order to prevent violations of causality (see "Detecting Concurrent Writes").
- It is possible for two nodes to independently generate writes with the same timestamp, especially when the clock only has millisecond resolution. An additional tiebreaker value (which can simply be a large random number) is required to resolve such conflicts, but this approach can also lead to violations of causality [62].

Thus, even though it is tempting to resolve conflicts by keeping the most "recent" value and discarding others, it's important to be aware that the definition of "recent" depends on a local time-of-day clock, which may well be incorrect. Even with tightly NTP-synchronized clocks, you could send a packet at timestamp 100 ms (according to the sender's clock) and have it arrive at timestamp 99 ms (according to the recipient's clock)—so it appears as though the packet arrived before it was sent, which is impossible.

Could NTP synchronization be made accurate enough that such incorrect orderings cannot occur? Probably not, because NTP's synchronization accuracy is itself limited by the network round-trip time, in addition to other sources of error such as quartz drift. To guarantee a correct ordering, you would need the clock error to be significantly lower than the network delay, which is not possible.

So-called *logical clocks* [66], which are based on incrementing counters rather than an oscillating quartz crystal, are a safer alternative for ordering events (see "Detecting Concurrent Writes"). Logical clocks do not measure the time of day or the number of seconds elapsed, only the relative ordering of events (whether one event happened before or after another). In contrast, time-of-day

and monotonic clocks, which measure actual elapsed time, are also known as *physical clocks*. We'll look at logical clocks in more detail in "ID Generators and Logical Clocks".

Clock readings with a confidence interval

You may be able to read a machine's time-of-day clock with microsecond or even nanosecond resolution. But even if you can get such a fine-grained measurement, that doesn't mean the value is actually accurate to such precision. In fact, it most likely is not—as mentioned previously, the drift in an imprecise quartz clock can easily be several milliseconds, even if you synchronize with an NTP server on the local network every minute. With an NTP server on the public internet, the best possible accuracy is probably to the tens of milliseconds, and the error may easily spike to over 100 ms when there is network congestion.

Thus, it doesn't make sense to think of a clock reading as a point in time—it is more like a range of times, within a confidence interval: for example, a system may be 95% confident that the time now is between 10.3 and 10.5 seconds past the minute, but it doesn't know any more precisely than that [67]. If we only know the time +/– 100 ms, the microsecond digits in the timestamp are essentially meaningless.

The uncertainty bound can be calculated based on your time source. If you have a GPS receiver or atomic clock directly attached to your computer, the expected error range is determined by the device and, in the case of GPS, by the quality of the signal from the satellites. If you're getting the time from a server, the uncertainty is based on the expected quartz drift since your last sync with the server, plus the NTP server's uncertainty, plus the network round-trip time to the server (to a first approximation, and assuming you trust the server).

Unfortunately, most systems don't expose this uncertainty: for example, when you call `clock_gettime()`, the return value doesn't tell you the expected error of the timestamp, so you don't know if its confidence interval is five milliseconds or five years.

There are exceptions: the *TrueTime* API in Google's Spanner [45] and Amazon's ClockBound explicitly report the confidence interval on the local clock. When you ask it for the current time, you get back two values:

`[earliest, latest]`, which are the *earliest possible* and the *latest possible* timestamp. Based on its uncertainty calculations, the clock knows that the actual current time is somewhere within that interval. The width of the interval depends, among other things, on how long it has been since the local quartz clock was last synchronized with a more accurate clock source.

Synchronized clocks for global snapshots

In "Snapshot Isolation and Repeatable Read" we discussed *multi-version concurrency control* (MVCC), which is a very useful feature in databases that need to support both small, fast read-write transactions and large, long-running read-only transactions (e.g., for backups or analytics). It allows read-only transactions to see a *snapshot* of the database, a consistent state at a particular point in time, without locking and interfering with read-write transactions.

Generally, MVCC requires a monotonically increasing transaction ID. If a write happened later than the snapshot (i.e., the write has a greater transaction ID than the snapshot), that write is invisible to the snapshot transaction. On a single-node database, a simple counter is sufficient for generating transaction IDs.

However, when a database is distributed across many machines, potentially in multiple datacenters, a global, monotonically increasing transaction ID (across all shards) is difficult to generate, because it requires coordination. The transaction ID must reflect causality: if transaction B reads or overwrites a value that was previously written by transaction A, then B must have a higher transaction ID than A—otherwise, the snapshot would not be consistent. With lots of small, rapid transactions, creating transaction IDs in a distributed system becomes an untenable bottleneck. (We will discuss such ID generators in "ID Generators and Logical Clocks".)

Can we use the timestamps from synchronized time-of-day clocks as transaction IDs? If we could get the synchronization good enough, they would have the right properties: later transactions have a higher timestamp. The problem, of course, is the uncertainty about clock accuracy.

Spanner implements snapshot isolation across datacenters in this way [68, 69]. It uses the clock's confidence interval as reported by the TrueTime API, and is based on the following observation: if you have two confidence intervals, each

consisting of an earliest and latest possible timestamp ($A = [A_{earliest}, A_{latest}]$ and $B = [B_{earliest}, B_{latest}]$), and those two intervals do not overlap (i.e., $A_{earliest} < A_{latest} < B_{earliest} < B_{latest}$), then B definitely happened after A—there can be no doubt. Only if the intervals overlap are we unsure in which order A and B happened.

In order to ensure that transaction timestamps reflect causality, Spanner deliberately waits for the length of the confidence interval before committing a read-write transaction. By doing so, it ensures that any transaction that may read the data is at a sufficiently later time, so their confidence intervals do not overlap. In order to keep the wait time as short as possible, Spanner needs to keep the clock uncertainty as small as possible; for this purpose, Google deploys a GPS receiver or atomic clock in each datacenter, allowing clocks to be synchronized to within about 7 ms [45].

The atomic clocks and GPS receivers are not strictly necessary in Spanner: the important thing is to have a confidence interval, and the accurate clock sources only help keep that interval small. Other systems are beginning to adopt similar approaches: for example, YugabyteDB can leverage ClockBound when running on AWS [70], and several other systems now also rely on clock synchronization to various degrees [71, 72].

Process Pauses

Let's consider another example of dangerous clock use in a distributed system. Say you have a database with a single leader per shard. Only the leader is allowed to accept writes. How does a node know that it is still leader (that it hasn't been declared dead by the others), and that it may safely accept writes?

One option is for the leader to obtain a *lease* from the other nodes, which is similar to a lock with a timeout [73]. Only one node can hold the lease at any one time—thus, when a node obtains a lease, it knows that it is the leader for some amount of time, until the lease expires. In order to remain leader, the node must periodically renew the lease before it expires. If the node fails, it stops renewing the lease, so another node can take over when it expires.

You can imagine the request-handling loop looking something like this:

```
while (true) {
    request = getIncomingRequest();
```

```
        // Ensure that the lease always has at least 10 sec
        if (lease.expiryTimeMillis - System.currentTimeMill
            lease = lease.renew();
        }

        if (lease.isValid()) {
            process(request);
        }
    }
```

What's wrong with this code? Firstly, it's relying on synchronized clocks: the expiry time on the lease is set by a different machine (where the expiry may be calculated as the current time plus 30 seconds, for example), and it's being compared to the local system clock. If the clocks are out of sync by more than a few seconds, this code will start doing strange things.

Secondly, even if we change the protocol to only use the local monotonic clock, there is another problem: the code assumes that very little time passes between the point that it checks the time (`System.currentTimeMillis()`) and the time when the request is processed (`process(request)`). Normally this code runs very quickly, so the 10 second buffer is more than enough to ensure that the lease doesn't expire in the middle of processing a request.

However, what if there is an unexpected pause in the execution of the program? For example, imagine the thread stops for 15 seconds around the line `lease.isValid()` before finally continuing. In that case, it's likely that the lease will have expired by the time the request is processed, and another node has already taken over as leader. However, there is nothing to tell this thread that it was paused for so long, so this code won't notice that the lease has expired until the next iteration of the loop—by which time it may have already done something unsafe by processing the request.

Is it reasonable to assume that a thread might be paused for so long? Unfortunately yes. There are various reasons why this could happen:

- Contention among threads accessing a shared resource, such as a lock or queue, can cause threads to spend a lot of their time waiting. Moving to a machine with more CPU cores can make such problems worse, and contention problems can be difficult to diagnose [74].

- Many programming language runtimes (such as the Java Virtual Machine) have a *garbage collector* (GC) that occasionally needs to stop all running threads. In the past, such *"stop-the-world" GC pauses* would sometimes last for several minutes [75]! With modern GC algorithms this is less of a problem, but GC pauses can still be noticable (see "Limiting the impact of garbage collection").
- In virtualized environments, a virtual machine can be *suspended* (pausing the execution of all processes and saving the contents of memory to disk) and *resumed* (restoring the contents of memory and continuing execution). This pause can occur at any time in a process's execution and can last for an arbitrary length of time. This feature is sometimes used for *live migration* of virtual machines from one host to another without a reboot, in which case the length of the pause depends on the rate at which processes are writing to memory [76].
- On end-user devices such as laptops and phones, execution may also be suspended and resumed arbitrarily, e.g., when the user closes the lid of their laptop.
- When the operating system context-switches to another thread, or when the hypervisor switches to a different virtual machine (when running in a virtual machine), the currently running thread can be paused at any arbitrary point in the code. In the case of a virtual machine, the CPU time spent in other virtual machines is known as *steal time*. If the machine is under heavy load—i.e., if there is a long queue of threads waiting to run—it may take some time before the paused thread gets to run again.
- If the application performs synchronous disk access, a thread may be paused waiting for a slow disk I/O operation to complete [77]. In many languages, disk access can happen surprisingly, even if the code doesn't explicitly mention file access—for example, the Java classloader lazily loads class files when they are first used, which could happen at any time in the program execution. I/O pauses and GC pauses may even conspire to combine their delays [78]. If the disk is actually a network filesystem or network block device (such as Amazon's EBS), the I/O latency is further subject to the variability of network delays [31].
- If the operating system is configured to allow *swapping to disk* (*paging*), a simple memory access may result in a page fault that requires a page from disk to be loaded into memory. The thread is paused while this slow I/O operation takes place. If memory pressure is high, this may in turn require a different page to be swapped out to disk. In extreme circumstances, the operating system may spend most of its time swapping pages in and out of

memory and getting little actual work done (this is known as *thrashing*). To avoid this problem, paging is often disabled on server machines (if you would rather kill a process to free up memory than risk thrashing).
- A Unix process can be paused by sending it the `SIGSTOP` signal, for example by pressing Ctrl-Z in a shell. This signal immediately stops the process from getting any more CPU cycles until it is resumed with `SIGCONT`, at which point it continues running where it left off. Even if your environment does not normally use `SIGSTOP`, it might be sent accidentally by an operations engineer.

All of these occurrences can *preempt* the running thread at any point and resume it at some later time, without the thread even noticing. The problem is similar to making multi-threaded code on a single machine thread-safe: you can't assume anything about timing, because arbitrary context switches and parallelism may occur.

When writing multi-threaded code on a single machine, we have fairly good tools for making it thread-safe: mutexes, semaphores, atomic counters, lock-free data structures, blocking queues, and so on. Unfortunately, these tools don't directly translate to distributed systems, because a distributed system has no shared memory—only messages sent over an unreliable network.

A node in a distributed system must assume that its execution can be paused for a significant length of time at any point, even in the middle of a function. During the pause, the rest of the world keeps moving and may even declare the paused node dead because it's not responding. Eventually, the paused node may continue running, without even noticing that it was asleep until it checks its clock sometime later.

Response time guarantees

In many programming languages and operating systems, threads and processes may pause for an unbounded amount of time, as discussed. Those reasons for pausing *can* be eliminated if you try hard enough.

Some software runs in environments where a failure to respond within a specified time can cause serious damage: computers that control aircraft, rockets, robots, cars, and other physical objects must respond quickly and predictably to their sensor inputs. In these systems, there is a specified *deadline* by which the software must respond; if it doesn't meet the deadline,

that may cause a failure of the entire system. These are so-called *hard real-time* systems.

NOTE

In embedded systems, *real-time* means that a system is carefully designed and tested to meet specified timing guarantees in all circumstances. This meaning is in contrast to the more vague use of the term *real-time* on the web, where it describes servers pushing data to clients and stream processing without hard response time constraints (see Chapter 12).

For example, if your car's onboard sensors detect that you are currently experiencing a crash, you wouldn't want the release of the airbag to be delayed due to an inopportune GC pause in the airbag release system.

Providing real-time guarantees in a system requires support from all levels of the software stack: a *real-time operating system* (RTOS) that allows processes to be scheduled with a guaranteed allocation of CPU time in specified intervals is needed; library functions must document their worst-case execution times; dynamic memory allocation may be restricted or disallowed entirely (real-time garbage collectors exist, but the application must still ensure that it doesn't give the GC too much work to do); and an enormous amount of testing and measurement must be done to ensure that guarantees are being met.

All of this requires a large amount of additional work and severely restricts the range of programming languages, libraries, and tools that can be used (since most languages and tools do not provide real-time guarantees). For these reasons, developing real-time systems is very expensive, and they are most commonly used in safety-critical embedded devices. Moreover, "real-time" is not the same as "high-performance"—in fact, real-time systems may have lower throughput, since they have to prioritize timely responses above all else (see also "Latency and Resource Utilization").

For most server-side data processing systems, real-time guarantees are simply not economical or appropriate. Consequently, these systems must suffer the pauses and clock instability that come from operating in a non-real-time environment.

Limiting the impact of garbage collection

Garbage collection used to be one of the biggest reasons for process pauses [79], but fortunately GC algorithms have improved a lot: a properly tuned collector will now usually pause for no more than a few milliseconds. The Java runtime offers collectors such as concurrent mark sweep (CMS), garbage-first (G1), the Z garbage collector (ZGC), Epsilon, and Shenandoah. Each of these is optimized for different memory profiles such as high-frequency object creation, large heaps, and so on. By contrast, Go offers a simpler concurrent mark sweep garbage collector that attempts to optimize itself.

If you need to avoid GC pauses entirely, one option is to use a language that doesn't have a garbage collector at all. For example, Swift uses automatic reference counting to determine when memory can be freed; Rust and Mojo track lifetimes of objects using the type system so the compiler can determine how long memory must be allocated for.

It's also possible to use a garbage-collected language while mitigating the impact of pauses. For example, objects can be stored and re-used in pools rather than discarded, or data can be allocated off-heap. A more extreme approach is to treat GC pauses like brief planned outages of a node, and to let other nodes handle requests from clients while one node is collecting its garbage. If the runtime can warn the application that a node soon requires a GC pause, the application can stop sending new requests to that node, wait for it to finish processing outstanding requests, and then perform the GC while no requests are in progress. This trick hides GC pauses from clients and reduces the high percentiles of the response time [80, 81].

A variant of this idea is to use the garbage collector only for short-lived objects (which are fast to collect) and to restart processes periodically, before they accumulate enough long-lived objects to require a full GC of long-lived objects [79, 82]. One node can be restarted at a time, and traffic can be shifted away from the node before the planned restart, like in a rolling upgrade (see Chapter 5).

These measures cannot fully prevent garbage collection pauses, but they can usefully reduce their impact on the application.

Knowledge, Truth, and Lies

So far in this chapter we have explored the ways in which distributed systems are different from programs running on a single computer: there is no shared memory, only message passing via an unreliable network with variable delays, and the systems may suffer from partial failures, unreliable clocks, and processing pauses.

The consequences of these issues are profoundly disorienting if you're not used to distributed systems. A node in the network cannot *know* anything for sure about other nodes—it can only make guesses based on the messages it receives (or doesn't receive). A node can only find out what state another node is in (what data it has stored, whether it is correctly functioning, etc.) by exchanging messages with it. If a remote node doesn't respond, there is no way of knowing what state it is in, because problems in the network cannot reliably be distinguished from problems at a node.

Discussions of these systems border on the philosophical: What do we know to be true or false in our system? How sure can we be of that knowledge, if the mechanisms for perception and measurement are unreliable [83]? Should software systems obey the laws that we expect of the physical world, such as cause and effect?

Fortunately, we don't need to go as far as figuring out the meaning of life. In a distributed system, we can state the assumptions we are making about the behavior (the *system model*) and design the actual system in such a way that it meets those assumptions. Algorithms can be proved to function correctly within a certain system model. This means that reliable behavior is achievable, even if the underlying system model provides very few guarantees.

However, although it is possible to make software well behaved in an unreliable system model, it is not straightforward to do so. In the rest of this chapter we will further explore the notions of knowledge and truth in distributed systems, which will help us think about the kinds of assumptions we can make and the guarantees we may want to provide. In Chapter 10 we will proceed to look at some examples of distributed algorithms that provide particular guarantees under particular assumptions.

The Majority Rules

Imagine a network with an asymmetric fault: a node is able to receive all messages sent to it, but any outgoing messages from that node are dropped or delayed [22]. Even though that node is working perfectly well, and is receiving requests from other nodes, the other nodes cannot hear its responses. After some timeout, the other nodes declare it dead, because they haven't heard from the node. The situation unfolds like a nightmare: the semi-disconnected node is dragged to the graveyard, kicking and screaming "I'm not dead!"—but since nobody can hear its screaming, the funeral procession continues with stoic determination.

In a slightly less nightmarish scenario, the semi-disconnected node may notice that the messages it is sending are not being acknowledged by other nodes, and so realize that there must be a fault in the network. Nevertheless, the node is wrongly declared dead by the other nodes, and the semi-disconnected node cannot do anything about it.

As a third scenario, imagine a node that pauses execution for one minute. During that time, no requests are processed and no responses are sent. The other nodes wait, retry, grow impatient, and eventually declare the node dead and load it onto the hearse. Finally, the pause finishes and the node's threads continue as if nothing had happened. The other nodes are surprised as the supposedly dead node suddenly raises its head out of the coffin, in full health, and starts cheerfully chatting with bystanders. At first, the paused node doesn't even realize that an entire minute has passed and that it was declared dead—from its perspective, hardly any time has passed since it was last talking to the other nodes.

The moral of these stories is that a node cannot necessarily trust its own judgment of a situation. A distributed system cannot exclusively rely on a single node, because a node may fail at any time, potentially leaving the system stuck and unable to recover. Instead, many distributed algorithms rely on a *quorum*, that is, voting among the nodes (see "Quorums for reading and writing"): decisions require some minimum number of votes from several nodes in order to reduce the dependence on any one particular node.

That includes decisions about declaring nodes dead. If a quorum of nodes declares another node dead, then it must be considered dead, even if that node

still very much feels alive. The individual node must abide by the quorum decision and step down.

Most commonly, the quorum is an absolute majority of more than half the nodes (although other kinds of quorums are possible). A majority quorum allows the system to continue working if a minority of nodes are faulty (with three nodes, one faulty node can be tolerated; with five nodes, two faulty nodes can be tolerated). However, it is still safe, because there can only be only one majority in the system—there cannot be two majorities with conflicting decisions at the same time. We will discuss the use of quorums in more detail when we get to *consensus algorithms* in Chapter 10.

Distributed Locks and Leases

Locks and leases in distributed application are prone to be misused, and a common source of bugs [84]. Let's look at one particular case of how they can go wrong.

In "Process Pauses" we saw that a lease is a kind of lock that times out and can be assigned to a new owner if the old owner stops responding (perhaps because it crashed, it paused for too long, or it was disconnected from the network). You can use leases in situations where a system requires there to be only one of some thing. For example:

- Only one node is allowed to be the leader for a database shard, to avoid split brain (see "Handling Node Outages").
- Only one transaction or client is allowed to update a particular resource or object, to prevent it being corrupted by concurrent writes.
- Only one node should process a given input file to a big processing job, to avoid wasted effort due to multiple nodes redundantly doing the same work.

It is worth thinking carefully about what happens if several nodes simultaneously believe that they hold the lease, perhaps due to a process pause. In the third example, the consequence is only some wasted computational resources, which is not a big deal. But in the first two cases, the consequence could be lost or corrupted data, which is much more serious.

For example, Figure 9-4 shows a data corruption bug due to an incorrect implementation of locking. (The bug is not theoretical: HBase used to have

this problem [85, 86].) Say you want to ensure that a file in a storage service can only be accessed by one client at a time, because if multiple clients tried to write to it, the file would become corrupted. You try to implement this by requiring a client to obtain a lease from a lock service before accessing the file. Such a lock service is often implemented using a consensus algorithm; we will discuss this further in Chapter 10.

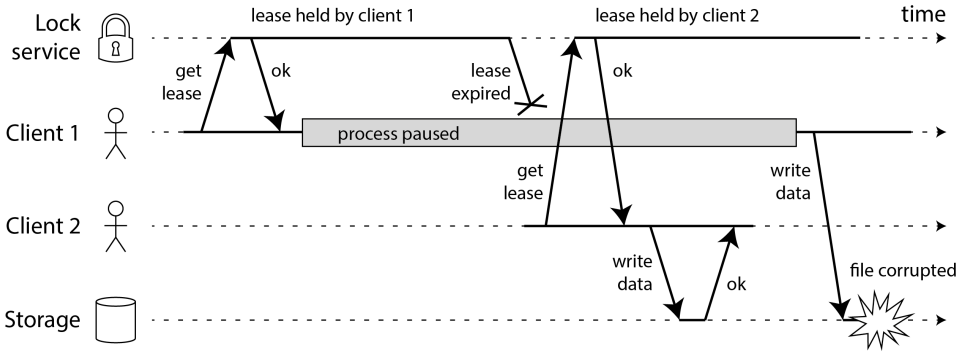

Figure 9-4. Incorrect implementation of a distributed lock: client 1 believes that it still has a valid lease, even though it has expired, and thus corrupts a file in storage.

The problem is an example of what we discussed in "Process Pauses": if the client holding the lease is paused for too long, its lease expires. Another client can obtain a lease for the same file, and start writing to the file. When the paused client comes back, it believes (incorrectly) that it still has a valid lease and proceeds to also write to the file. We now have a split brain situation: the clients' writes clash and corrupt the file.

Figure 9-5 shows a different problem that has similar consequences. In this example there is no process pause, only a crash by client 1. Just before client 1 crashes it sends a write request to the storage service, but this request is delayed for a long time in the network. (Remember from "Network Faults in Practice" that packets can sometimes be delayed by a minute or more.) By the time the write request arrives at the storage service, the lease has already timed out, allowing client 2 to acquire it and issue a write of its own. The result is corruption similar to Figure 9-4.

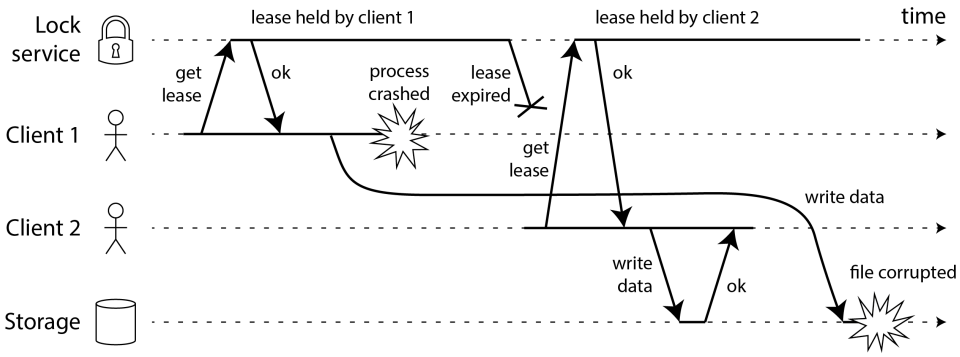

Figure 9-5. A message from a former leaseholder might be delayed for a long time, and arrive after another node has taken over the lease.

Fencing off zombies and delayed requests

The term *zombie* is sometimes used to describe a former leaseholder who has not yet found out that it lost the lease, and who is still acting as if it was the current leaseholder. Since we cannot rule out zombies entirely, we have to instead ensure that they can't do any damage in the form of split brain. This is called *fencing off* the zombie.

Some systems attempt to fence off zombies by shutting them down, for example by disconnecting them from the network [9], shutting down the VM via the cloud provider's management interface, or even physically powering down the machine [87]. This approach is known as *Shoot The Other Node In The Head* or STONITH. Unfortunately, it suffers from some problems: it does not protect against large network delays like in Figure 9-5; it can happen that all of the nodes shut each other down [19]; and by the time the zombie has been detected and shut down, it may already be too late and data may already have been corrupted.

A more robust fencing solution, which protects against both zombies and delayed requests, is illustrated in Figure 9-6.

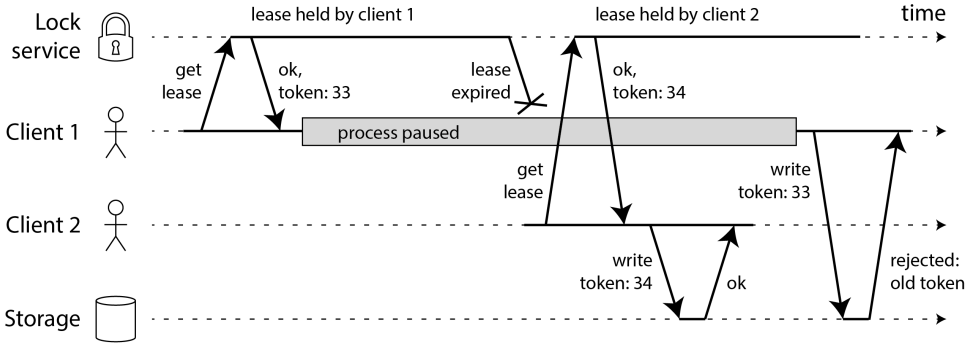

Figure 9-6. Making access to storage safe by allowing writes only in the order of increasing fencing tokens.

Let's assume that every time the lock service grants a lock or lease, it also returns a *fencing token*, which is a number that increases every time a lock is granted (e.g., incremented by the lock service). We can then require that every time a client sends a write request to the storage service, it must include its current fencing token.

NOTE

There are several alternative names for fencing tokens. In Chubby, Google's lock service, they are called *sequencers* [88], and in Kafka they are called *epoch numbers*. In consensus algorithms, which we will discuss in Chapter 10, the *ballot number* (Paxos) or *term number* (Raft) serves a similar purpose.

In Figure 9-6, client 1 acquires the lease with a token of 33, but then it goes into a long pause and the lease expires. Client 2 acquires the lease with a token of 34 (the number always increases) and then sends its write request to the storage service, including the token of 34. Later, client 1 comes back to life and sends its write to the storage service, including its token value 33. However, the storage service remembers that it has already processed a write with a higher token number (34), and so it rejects the request with token 33. A client that has just acquired the lease must immediately make a write to the storage service, and once that write has completed, any zombies are fenced off. These operations are similar to the optimistic concurrency control (OCC) technique we saw in "Pessimistic versus optimistic concurrency control"), except that fencing is permanent while concurrency control failures can be retried.

If ZooKeeper is your lock service, you can use the transaction ID `zxid` or the node version `cversion` as fencing token [85]. With etcd, the revision number along with the lease ID serves a similar purpose [89]. The FencedLock API in Hazelcast explicitly generates a fencing token [90].

This mechanism requires that the storage service has some way of checking whether a write is based on an outdated token. Alternatively, it's sufficient for the service to support a write that succeeds only if the object has not been written by another client since the current client last read it, similarly to an atomic compare-and-set (CAS) operation. For example, object storage services support such a check: Amazon S3 calls it *conditional writes*, Azure

Blob Storage calls it *conditional headers*, and Google Cloud Storage calls it *request preconditions*.

Fencing with multiple replicas

If your clients need to write only to one storage service that supports such conditional writes, the lock service is somewhat redundant [91, 92], since the lease assignment could have been implemented directly based on that storage service [93]. However, once you have a fencing token you can also use it with multiple services or replicas, and ensure that the old leaseholder is fenced off on all of those services.

For example, imagine the storage service is a leaderless replicated key-value store with last-write-wins conflict resolution (see "Leaderless Replication"). In such a system, the client sends writes directly to each replica, and each replica independently decides whether to accept a write based on a timestamp assigned by the client.

As illustrated in Figure 9-7, you can put the writer's fencing token in the most significant bits or digits of the timestamp. You can then be sure that any timestamp generated by the new leaseholder will be greater than any timestamp from the old leaseholder, even if the old leaseholder's writes happened later.

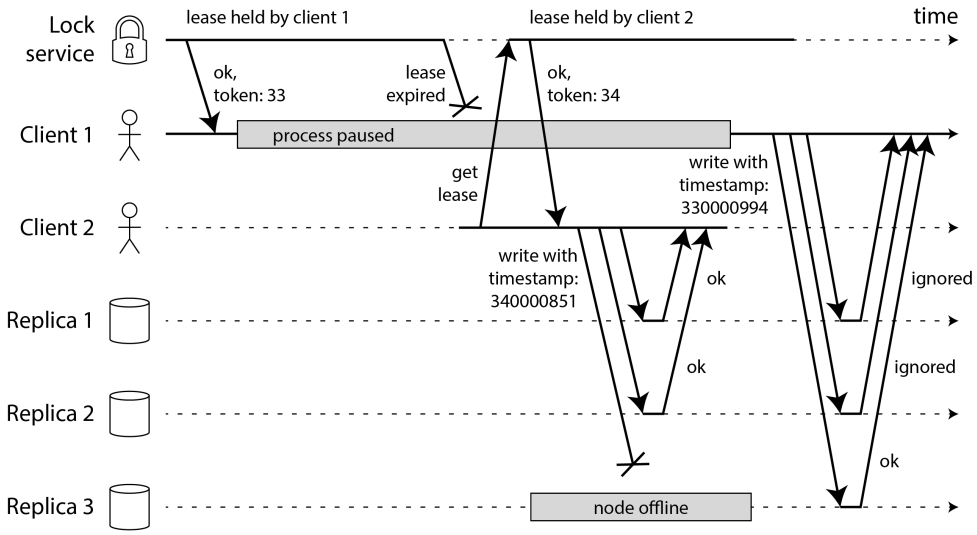

Figure 9-7. Using fencing tokens to protect writes to a leaderless replicated database.

In Figure 9-7, Client 2 has a fencing token of 34, so all of its timestamps starting with 34… are greater than any timestamps starting with 33… that are generated by Client 1. Client 2 writes to a quorum of replicas but it can't reach Replica 3. This means that when the zombie Client 1 later tries to write, its

write may succeed at Replica 3 even though it is ignored by replicas 1 and 2. This is not a problem, since a subsequent quorum read will prefer the write from Client 2 with the greater timestamp, and read repair or anti-entropy will eventually overwrite the value written by Client 1.

As you can see from these examples, it is not safe to assume that there is only one node holding a lease at any one time. Fortunately, with a bit of care you can use fencing tokens to prevent zombies and delayed requests from doing any damage.

Byzantine Faults

Fencing tokens can detect and block a node that is *inadvertently* acting in error (e.g., because it hasn't yet found out that its lease has expired). However, if the node deliberately wanted to subvert the system's guarantees, it could easily do so by sending messages with a fake fencing token.

In this book we assume that nodes are unreliable but honest: they may be slow or never respond (due to a fault), and their state may be outdated (due to a GC pause or network delays), but we assume that if a node *does* respond, it is telling the "truth": to the best of its knowledge, it is playing by the rules of the protocol.

Distributed systems problems become much harder if there is a risk that nodes may "lie" (send arbitrary faulty or corrupted responses)—for example, it might cast multiple contradictory votes in the same election. Such behavior is known as a *Byzantine fault*, and the problem of reaching consensus in this untrusting environment is known as the *Byzantine Generals Problem* [94].

THE BYZANTINE GENERALS PROBLEM

The Byzantine Generals Problem is a generalization of the so-called *Two Generals Problem* [95], which imagines a situation in which two army generals need to agree on a battle plan. As they have set up camp on two different sites, they can only communicate by messenger, and the messengers sometimes get delayed or lost (like packets in a network). We will discuss this problem of *consensus* in Chapter 10.

In the Byzantine version of the problem, there are n generals who need to agree, and their endeavor is hampered by the fact that there are some traitors in their midst. Most of the generals are loyal, and thus send truthful messages, but the traitors may try to deceive and confuse the others by sending fake or untrue messages. It is not known in advance who the traitors are.

Byzantium was an ancient Greek city that later became Constantinople, in the place which is now Istanbul in Turkey. There isn't any historic evidence that the generals of Byzantium were any more prone to intrigue and conspiracy than those elsewhere. Rather, the name is derived from *Byzantine* in the sense of *excessively complicated, bureaucratic, devious*, which was used in politics long before computers [96]. Lamport wanted to choose a nationality that would not offend any readers, and he was advised that calling it *The Albanian Generals Problem* was not such a good idea [97].

A system is *Byzantine fault-tolerant* if it continues to operate correctly even if some of the nodes are malfunctioning and not obeying the protocol, or if malicious attackers are interfering with the network. This concern is relevant in certain specific circumstances. For example:

- In aerospace environments, the data in a computer's memory or CPU register could become corrupted by radiation, leading it to respond to other nodes in arbitrarily unpredictable ways. Since a system failure would be very expensive (e.g., an aircraft crashing and killing everyone on board, or a rocket colliding with the International Space Station), flight control systems must tolerate Byzantine faults [98, 99].
- In a system with multiple participating parties, some participants may attempt to cheat or defraud others. In such circumstances, it is not safe for a node to simply trust another node's messages, since they may be sent with malicious intent. For example, cryptocurrencies like Bitcoin and

other blockchains can be considered to be a way of getting mutually untrusting parties to agree whether a transaction happened or not, without relying on a central authority [100].

However, in the kinds of systems we discuss in this book, we can usually safely assume that there are no Byzantine faults. In a datacenter, all the nodes are controlled by your organization (so they can hopefully be trusted) and radiation levels are low enough that memory corruption is not a major problem (although datacenters in orbit are being considered [101]). Multitenant systems have mutually untrusting tenants, but they are isolated from each other using firewalls, virtualization, and access control policies, not using Byzantine fault tolerance. Protocols for making systems Byzantine fault-tolerant are quite expensive [102], and fault-tolerant embedded systems rely on support from the hardware level [98]. In most server-side data systems, the cost of deploying Byzantine fault-tolerant solutions makes them impracticable.

Web applications do need to expect arbitrary and malicious behavior of clients that are under end-user control, such as web browsers. This is why input validation, sanitization, and output escaping are so important: to prevent SQL injection and cross-site scripting, for example. However, we typically don't use Byzantine fault-tolerant protocols here, but simply make the server the authority on deciding what client behavior is and isn't allowed. In peer-to-peer networks, where there is no such central authority, Byzantine fault tolerance is more relevant [103, 104].

A bug in the software could be regarded as a Byzantine fault, but if you deploy the same software to all nodes, then a Byzantine fault-tolerant algorithm cannot save you. Most Byzantine fault-tolerant algorithms require a supermajority of more than two-thirds of the nodes to be functioning correctly (for example, if you have four nodes, at most one may malfunction). To use this approach against bugs, you would have to have four independent implementations of the same software and hope that a bug only appears in one of the four implementations.

Similarly, it would be appealing if a protocol could protect us from vulnerabilities, security compromises, and malicious attacks. Unfortunately, this is not realistic either: in most systems, if an attacker can compromise one node, they can probably compromise all of them, because they are probably running the same software. Thus, traditional mechanisms (authentication,

access control, encryption, firewalls, and so on) continue to be the main protection against attackers.

Weak forms of lying

Although we assume that nodes are generally honest, it can be worth adding mechanisms to software that guard against weak forms of "lying"—for example, invalid messages due to hardware issues, software bugs, and misconfiguration. Such protection mechanisms are not full-blown Byzantine fault tolerance, as they would not withstand a determined adversary, but they are nevertheless simple and pragmatic steps toward better reliability. For example:

- Network packets do sometimes get corrupted due to hardware issues or bugs in operating systems, drivers, routers, etc. Usually, corrupted packets are caught by the checksums built into TCP and UDP, but sometimes they evade detection [105, 106, 107]. Simple measures are usually sufficient protection against such corruption, such as checksums in the application-level protocol. TLS-encrypted connections also offer protection against corruption.
- A publicly accessible application must carefully sanitize any inputs from users, for example escaping certain characters to prevent SQL injection attacks, checking that a value is within a reasonable range, and limiting the size of strings to prevent denial of service through large memory allocations. An internal service behind a firewall may be able to get away with less strict checks on inputs, but basic checks in protocol parsers are still a good idea [105].
- NTP clients can be configured with multiple server addresses. When synchronizing, the client contacts all of them, estimates their errors, and checks that a majority of servers agree on some time range. As long as most of the servers are okay, a misconfigured NTP server that is reporting an incorrect time is detected as an outlier and is excluded from synchronization [39]. The use of multiple servers makes NTP more robust than if it only uses a single server.

System Model and Reality

Many algorithms have been designed to solve distributed systems problems—for example, we will examine solutions for the consensus problem in

Chapter 10. In order to be useful, these algorithms need to tolerate the various faults of distributed systems that we discussed in this chapter.

Algorithms need to be written in a way that does not depend too heavily on the details of the hardware and software configuration on which they are run. This in turn requires that we somehow formalize the kinds of faults that we expect to happen in a system. We do this by defining a *system model*, which is an abstraction that describes what things an algorithm may assume.

With regard to timing assumptions, three system models are in common use:

Synchronous model

The synchronous model assumes bounded network delay, bounded process pauses, and bounded clock error. This does not imply exactly synchronized clocks or zero network delay; it just means you know that network delay, pauses, and clock drift will never exceed some fixed upper bound [108]. The synchronous model is not a realistic model of most practical systems, because (as discussed in this chapter) unbounded delays and pauses do occur.

Partially synchronous model

Partial synchrony means that a system behaves like a synchronous system *most of the time*, but it sometimes exceeds the bounds for network delay, process pauses, and clock drift [108]. This is a realistic model of many systems: most of the time, networks and processes are quite well behaved—otherwise we would never be able to get anything done—but we have to reckon with the fact that any timing assumptions may be shattered occasionally. When this happens, network delay, pauses, and clock error may become arbitrarily large.

Asynchronous model

In this model, an algorithm is not allowed to make any timing assumptions—in fact, it does not even have a clock (so it cannot use timeouts). Some algorithms can be designed for the asynchronous model, but it is very restrictive.

Moreover, besides timing issues, we have to consider node failures. Some common system models for nodes are:

Crash-stop faults

In the *crash-stop* (or *fail-stop*) model, an algorithm may assume that a node can fail in only one way, namely by crashing [109]. This means that the node may suddenly stop responding at any moment, and thereafter that node is gone forever—it never comes back.

Crash-recovery faults

We assume that nodes may crash at any moment, and perhaps start responding again after some unknown time. In the crash-recovery model, nodes are assumed to have stable storage (i.e., nonvolatile disk storage) that is preserved across crashes, while the in-memory state is assumed to be lost.

Degraded performance and partial functionality

In addition to crashing and restarting, nodes may go slow: they may still be able to respond to health check requests, while being too slow to get any real work done. For example, a Gigabit network interface could suddenly drop to 1 Kb/s throughput due to a driver bug [110]; a process that is under memory pressure may spend most of its time performing garbage collection [111]; worn-out SSDs can have erratic performance; and hardware can be affected by high temperature, loose connectors, mechanical vibration, power supply problems, firmware bugs, and more [112]. Such a situation is called a *limping node*, *gray failure*, or *fail-slow* [113], and it can be even more difficult to deal with than a cleanly failed node. A related problem is when a process stops doing some of the things it is supposed to do while other aspects continue working, for example because a background thread is crashed or deadlocked [114].

Byzantine (arbitrary) faults

Nodes may do absolutely anything, including trying to trick and deceive other nodes, as described in the last section.

For modeling real systems, the partially synchronous model with crash-recovery faults is generally the most useful model. It allows for unbounded network delay, process pauses, and slow nodes. But how do distributed algorithms cope with that model?

Defining the correctness of an algorithm

To define what it means for an algorithm to be *correct*, we can describe its *properties*. For example, the output of a sorting algorithm has the property that for any two distinct elements of the output list, the element further to the left is smaller than the element further to the right. That is simply a formal way of defining what it means for a list to be sorted—an invariant of a sorted list.

Similarly, we can write down the properties we want of a distributed algorithm to define what it means to be correct. For example, if we are generating fencing tokens for a lock (see "Fencing off zombies and delayed requests"), we may require the algorithm to have the following properties:

Uniqueness

No two requests for a fencing token return the same value.

Monotonic sequence

If request x returned token t_x, and request y returned token t_y, and x completed before y began, then $t_x < t_y$.

Availability

A node that requests a fencing token and does not crash eventually receives a response.

An algorithm is correct in some system model if it always satisfies its properties in all situations that we assume may occur in that system model. However, if all nodes crash, or all network delays suddenly become infinitely long, then no algorithm will be able to get anything done. How can we still make useful guarantees even in a system model that allows complete failures?

Safety and liveness

To clarify the situation, it is worth distinguishing between two different kinds of properties: *safety* and *liveness* properties. In the example just given, *uniqueness* and *monotonic sequence* are safety properties, but *availability* is a liveness property.

What distinguishes the two kinds of properties? A giveaway is that liveness properties often include the word "eventually" in their definition. (And yes, you guessed it—*eventual consistency* is a liveness property [115].)

Safety is often informally defined as *nothing bad happens*, and liveness as *something good eventually happens*. However, it's best to not read too much into those informal definitions, because "good" and "bad" are value judgements that don't apply well to algorithms. The actual definitions of safety and liveness are more precise [116]:

- If a safety property is violated, we can point at a particular point in time at which it was broken (for example, if the uniqueness property was violated, we can identify the particular operation in which a duplicate fencing token was returned). After a safety property has been violated, the violation cannot be undone—the damage is already done.
- A liveness property works the other way round: it may not hold at some point in time (for example, a node may have sent a request but not yet received a response), but there is always hope that it may be satisfied in the future (namely by receiving a response).

An advantage of distinguishing between safety and liveness properties is that it helps us deal with difficult system models. For distributed algorithms, it is common to require that safety properties *always* hold, in all possible situations of a system model [108]. That is, even if all nodes crash, or the entire network fails, the algorithm must nevertheless ensure that it does not return a wrong result (i.e., that the safety properties remain satisfied).

However, with liveness properties we are allowed to make caveats: for example, we could say that a request needs to receive a response only if a majority of nodes have not crashed, and only if the network eventually recovers from an outage. The definition of the partially synchronous model requires that eventually the system returns to a synchronous state—that is, any period of network interruption lasts only for a finite duration and is then repaired.

Mapping system models to the real world

Safety and liveness properties and system models are very useful for reasoning about the correctness of a distributed algorithm. However, when implementing an algorithm in practice, the messy facts of reality come back to

bite you again, and it becomes clear that the system model is a simplified abstraction of reality.

For example, algorithms in the crash-recovery model generally assume that data in stable storage survives crashes. However, what happens if the data on disk is corrupted, or the data is wiped out due to hardware error or misconfiguration [117]? What happens if a server has a firmware bug and fails to recognize its hard drives on reboot, even though the drives are correctly attached to the server [118]?

Quorum algorithms (see "Quorums for reading and writing") rely on a node remembering the data that it claims to have stored. If a node may suffer from amnesia and forget previously stored data, that breaks the quorum condition, and thus breaks the correctness of the algorithm. Perhaps a new system model is needed, in which we assume that stable storage mostly survives crashes, but may sometimes be lost. But that model then becomes harder to reason about.

The theoretical description of an algorithm can declare that certain things are simply assumed not to happen—and in non-Byzantine systems, we do have to make some assumptions about faults that can and cannot happen. However, a real implementation may still have to include code to handle the case where something happens that was assumed to be impossible, even if that handling boils down to `printf("Sucks to be you")` and `exit(666)`—i.e., letting a human operator clean up the mess [119]. (This is one difference between computer science and software engineering.)

That is not to say that theoretical, abstract system models are worthless—quite the opposite. They are incredibly helpful for distilling down the complexity of real systems to a manageable set of faults that we can reason about, so that we can understand the problem and try to solve it systematically.

Formal Methods and Randomized Testing

How do we know that an algorithm satisfies the required properties? Due to concurrency, partial failures, and network delays there are a huge number of potential states. We need to guarantee that the properties hold in every possible state, and ensure that we haven't forgotten about any edge cases.

One approach is to formally verify an algorithm by describing it mathematically, and using proof techniques to show that it satisfies the

required properties in all situations that the system model allows. Proving an algorithm correct does not mean its *implementation* on a real system will necessarily always behave correctly. But it's a very good first step, because the theoretical analysis can uncover problems in an algorithm that might remain hidden for a long time in a real system, and that only come to bite you when your assumptions (e.g., about timing) are defeated due to unusual circumstances.

It is prudent to combine theoretical analysis with empirical testing to verify that implementations behave as expected. Techniques such as property-based testing, fuzzing, and deterministic simulation testing (DST) use randomization to test a system in a wide range of situations. Organizations such as Amazon Web Services, FoundationDB, and TigerBeetle have successfully used a combination of these techniques on many of their products [120, 121, 122, 123].

Model checking and specification languages

Model checkers are tools that help verify that an algorithm or system behaves as expected. An algorithm specification is written in a purpose-built language such as TLA+, Gallina, or FizzBee. These languages make it easier to focus on an algorithm's behavior without worrying about code implementation details. Model checkers then use these models to verify that invariants hold across all of an algorithm's states by systematically trying all the things that could happen.

Model checking can't actually prove that an algorithm's invariants hold for every possible state since most real-world algorithms have an infinite state space. A true verification of all states would require a formal proof, which can be done, but which is typically more difficult than running a model checker. Instead, model checkers encourage you to reduce the algorithm's model to an approximation that can be fully verified, or to limit the execution to some upper bound (for example, by setting a maximum number of messages that can be sent). Any bugs that only occur with longer executions would then not be found.

Still, model checkers strike a nice balance between ease of use and the ability to find non-obvious bugs. CockroachDB, TiDB, Kafka, and many other distributed systems use model specifications to find and fix bugs [124, 125, 126]. For example, using TLA+, researchers were able to demonstrate the

potential for data loss in viewstamped replication (VR) caused by ambiguity in the prose description of the algorithm [127].

By design, model checkers don't run your actual code, but rather a simplified model that specifies only the core ideas of your protocol. This makes it more tractable to systematically explore the state space, but it risks that your specification and your implementation go out of sync with each other [128]. It is possible to check whether the model and the real implementation have equivalent behavior, but this requires instrumentation in the real implementation [129].

Fault injection

Many bugs are triggered when machine and network failures occur. Fault injection is an effective (and sometimes scary) technique that verifies whether a system's implementation works as expected things go wrong. The idea is simple: inject faults into a running system's environment and see how it behaves. Faults can be network failures, machine crashes, disk corruption, paused processes—anything you can imagine going wrong with a computer.

Fault injection tests are typically run in an environment that closely resembles the production environment where the system will run. Some even inject faults directly into their production environment. Netflix popularized this approach with their Chaos Monkey tool [130]. Production fault injection is often referred to as *chaos engineering*, which we discussed in "Reliability and Fault Tolerance".

To run fault injection tests, the system under test is first deployed along with fault injection coordinators and scripts. Coordinators are responsible for deciding what faults to execute and when to execute them. Local or remote scripts are responsible for injecting failures into individual nodes or processes. Injection scripts use many different tools to trigger faults. A Linux process can be paused or killed using Linux's `kill` command, a disk can be unmounted with `umount`, and network connections can be disrupted through firewall settings. You can inspect system behavior during and after faults are injected to make sure things work as expected.

The myriad of tools required to trigger failures make fault injection tests cumbersome to write. It's common to adopt a fault injection framework like Jepsen to run fault injection tests to simplify the process. Such frameworks

come with integrations for various operating systems and many pre-built fault injectors [131]. Jepsen has been remarkably effective at finding critical bugs in many widely-used systems [132, 133].

Deterministic simulation testing

Deterministic simulation testing (DST) has also become a popular complement to model-checking and fault injection. It uses a similar state space exploration process as a model checker, but it tests your actual code, not a model.

In DST, a simulation automatically runs through a large number of randomised executions of the system. Network communication, I/O, and clock timing during the simulation are all replaced with mocks that allow the simulator to control the exact order in which things happen, including various timings and failure scenarios. This allows the simulator to explore many more situations than hand-written tests or fault injection could. If a test fails, it can be re-run since the simulator knows the exact order of operations that triggered the failure—in contrast to fault injection, which does not have such fine-grained control over the system.

DST requires the simulator to be able to control all sources of nondeterminism, such as network delays or thread scheduling in multithreaded code. One of three strategies is generally adopted to make code deterministic:

Application-level

Some systems are built from the ground-up to make it easy to execute code deterministically. For example, FoundationDB, one of the pioneers in the DST space, is built using an asynchronous communication library called Flow. Flow provides a point for developers to inject a deterministic network simulation into the system [134]. Similarly, TigerBeetle is an online transaction processing (OLTP) database with first-class DST support. The system's state is modeled as a state machine, with all mutations occuring within a single event loop. When combined with mock deterministic primitives such as clocks, such an architecture is able to run deterministically [135].

Runtime-level

Languages with asynchronous runtimes and commonly used libraries provide an insertion point to introduce determinism. A single-threaded runtime is used to force all asynchronous code to run sequentially. FrostDB, for example, patches Go's runtime to execute goroutines sequentially [136]. Rust's madsim library works in a similar manner. Madsim provides deterministic implementations of Tokio's asynchronous runtime API, AWS's S3 library, Kafka's Rust library, and many others. Applications can swap in deterministic libraries and runtimes to get deterministic test executions without changing their code.

Machine-level

Rather than patching code at runtime, an entire machine can be made deterministic. This is a delicate process that requires a machine to respond to all normally nondeterministic calls with deterministic responses. Tools such as Antithesis do this by building a custom hypervisor that replaces normally nondeterministic operations with deterministic ones. Everything from clocks to network and storage needs to be accounted for. Once done, though, developers can run their entire distributed system in a collection of containers within the hypervisor and get a completely deterministic distributed system.

DST provides several advantages beyond replayability. Tools such as Antithesis attempt to explore many different code paths in application code by branching a test execution into multiple sub-executions when it discovers less common behavior. And because deterministic tests often use mocked clocks and network calls, such tests can run faster than wall-clock time. For example, TigerBeetle's time abstraction allows simulations to simulate network latency and timeouts without actually taking the full length of time to trigger the timeout. Such techniques allow the simulator to explore more code paths faster.

THE POWER OF DETERMINISM

Nondeterminism is at the core of all of the distributed systems challenges we discussed in this chapter: concurrency, network delay, process pauses, clock jumps, and crashes all happen in unpredictable ways that vary from one run of a system to the next. Conversely, if you can make a system deterministic, that can hugely simplify things.

In fact, making things deterministic is a simple but powerful idea that arises again and again in distributed system design. Besides deterministic simulation testing, we have seen several ways of using determinism over the past chapters:

- A key advantage of event sourcing (see "Event Sourcing and CQRS") is that you can deterministically replay a log of events to reconstruct derived materialized views.
- Workflow engines (see "Durable Execution and Workflows") rely on workflow definitions being deterministic to provide durable execution semantics.
- *State machine replication*, which we will discuss in "Using shared logs", replicates data by independently executing the same sequence of deterministic transactions on each replica. We have already seen two variants of that idea: statement-based replication (see "Implementation of Replication Logs") and serial transaction execution using stored procedures (see "Pros and cons of stored procedures").

However, making code fully deterministic requires care. Even once you have removed all concurrency and replaced I/O, network communication, clocks, and random number generators with deterministic simulations, elements of nondeterminism may remain. For example, in some programming languages, the order in which you iterate over the elements of a hash table may be nondeterministic. Whether you run into a resource limit (memory allocation failure, stack overflow) is also nondeterministic.

Summary

In this chapter we have discussed a wide range of problems that can occur in distributed systems, including:

- Whenever you try to send a packet over the network, it may be lost or arbitrarily delayed. Likewise, the reply may be lost or delayed, so if you don't get a reply, you have no idea whether the message got through.
- A node's clock may be significantly out of sync with other nodes (despite your best efforts to set up NTP), it may suddenly jump forward or back in time, and relying on it is dangerous because you most likely don't have a good measure of your clock's confidence interval.
- A process may pause for a substantial amount of time at any point in its execution, be declared dead by other nodes, and then come back to life again without realizing that it was paused.

The fact that such *partial failures* can occur is the defining characteristic of distributed systems. Whenever software tries to do anything involving other nodes, there is the possibility that it may occasionally fail, or randomly go slow, or not respond at all (and eventually time out). In distributed systems, we try to build tolerance of partial failures into software, so that the system as a whole may continue functioning even when some of its constituent parts are broken.

To tolerate faults, the first step is to *detect* them, but even that is hard. Most systems don't have an accurate mechanism of detecting whether a node has failed, so most distributed algorithms rely on timeouts to determine whether a remote node is still available. However, timeouts can't distinguish between network and node failures, and variable network delay sometimes causes a node to be falsely suspected of crashing. Handling limping nodes, which are responding but are too slow to do anything useful, is even harder.

Once a fault is detected, making a system tolerate it is not easy either: there is no global variable, no shared memory, no common knowledge or any other kind of shared state between the machines [83]. Nodes can't even agree on what time it is, let alone on anything more profound. The only way information can flow from one node to another is by sending it over the unreliable network. Major decisions cannot be safely made by a single node, so we require protocols that enlist help from other nodes and try to get a quorum to agree.

If you're used to writing software in the idealized mathematical perfection of a single computer, where the same operation always deterministically returns the same result, then moving to the messy physical reality of distributed systems can be a bit of a shock. Conversely, distributed systems engineers will

often regard a problem as trivial if it can be solved on a single computer [4], and indeed a single computer can do a lot nowadays. If you can avoid opening Pandora's box and simply keep things on a single machine, for example by using an embedded storage engine (see "Embedded Storage Engines"), it is generally worth doing so.

However, as discussed in "Distributed Versus Single-Node Systems", scalability is not the only reason for wanting to use a distributed system. Fault tolerance and low latency (by placing data geographically close to users) are equally important goals, and those things cannot be achieved with a single node. The power of distributed systems is that in principle, they can run forever without being interrupted at the service level, because all faults and maintenance can be handled at the node level. (In practice, if a bad configuration change is rolled out to all nodes, that will still bring a distributed system to its knees.)

In this chapter we also went on some tangents to explore whether the unreliability of networks, clocks, and processes is an inevitable law of nature. We saw that it isn't: it is possible to give hard real-time response guarantees and bounded delays in networks, but doing so is very expensive and results in lower utilization of hardware resources. Most non-safety-critical systems choose cheap and unreliable over expensive and reliable.

This chapter has been all about problems, and has given us a bleak outlook. We gain a lot by using extensively tested, production-grade distributed systems that manage these problems. In the next chapter we will move on to solutions, and discuss some algorithms such systems employ to cope with these issues.

FOOTNOTES

REFERENCES

[1] Mark Cavage. There's Just No Getting Around It: You're Building a Distributed System. *ACM Queue*, volume 11, issue 4, pages 80-89, April 2013. doi:10.1145/2466486.2482856

[2] Jay Kreps. Getting Real About Distributed System Reliability. *blog.empathybox.com*, March 2012. Archived at perma.cc/9B5Q-AEBW

[3] Coda Hale. You Can't Sacrifice Partition Tolerance. *codahale.com*, October 2010. https://perma.cc/6GJU-X4G5

[4] Jeff Hodges. Notes on Distributed Systems for Young Bloods. *somethingsimilar.com*, January 2013. Archived at perma.cc/B636-62CE

[5] Van Jacobson. Congestion Avoidance and Control. At *ACM Symposium on Communications Architectures and Protocols* (SIGCOMM), August 1988. doi:10.1145/52324.52356

[6] Bert Hubert. The Ultimate SO_LINGER Page, or: Why Is My TCP Not Reliable. *blog.netherlabs.nl*, January 2009. Archived at perma.cc/6HDX-L2RR

[7] Jerome H. Saltzer, David P. Reed, and David D. Clark. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems*, volume 2, issue 4, pages 277–288, November 1984. doi:10.1145/357401.357402

[8] Peter Bailis and Kyle Kingsbury. The Network Is Reliable. *ACM Queue*, volume 12, issue 7, pages 48-55, July 2014. doi:10.1145/2639988.2639988

[9] Joshua B. Leners, Trinabh Gupta, Marcos K. Aguilera, and Michael Walfish. Taming Uncertainty in Distributed Systems with Help from the Network. At *10th European Conference on Computer Systems* (EuroSys), April 2015. doi:10.1145/2741948.2741976

[10] Phillipa Gill, Navendu Jain, and Nachiappan Nagappan. Understanding Network Failures in Data Centers: Measurement, Analysis, and Implications. At *ACM SIGCOMM Conference*, August 2011. doi:10.1145/2018436.2018477

[11] Urs Hölzle. But recently a farmer had started grazing a herd of cows nearby. And whenever they stepped on the fiber link, they bent it enough to cause a blip. *x.com*, May 2020. Archived at perma.cc/WX8X-ZZA5

[12] CBC News. Hundreds lose internet service in northern B.C. after beaver chews through cable. *cbc.ca*, April 2021. Archived at perma.cc/UW8C-H2MY

[13] Will Oremus. The Global Internet Is Being Attacked by Sharks, Google Confirms. *slate.com*, August 2014. Archived at perma.cc/P6F3-C6YG

[14] Jess Auerbach Jahajeeah. Down to the wire: The ship fixing our internet. *continent.substack.com*, November 2023. Archived at perma.cc/DP7B-EQ7S

[15] Santosh Janardhan. More details about the October 4 outage. *engineering.fb.com*, October 2021. Archived at perma.cc/WW89-VSXH

[16] Tom Parfitt. Georgian woman cuts off web access to whole of Armenia. *theguardian.com*, April 2011. Archived at perma.cc/KMC3-N3NZ

[17] Antonio Voce, Tural Ahmedzade and Ashley Kirk. 'Shadow fleets' and subaquatic sabotage: are Europe's undersea internet cables under attack? *theguardian.com*, March 2025. Archived at perma.cc/HA7S-ZDBV

[18] Shengyun Liu, Paolo Viotti, Christian Cachin, Vivien Quéma, and Marko Vukolić. XFT: Practical Fault Tolerance beyond Crashes. At *12th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), November 2016.

[19] Mark Imbriaco. Downtime last Saturday. *github.blog*, December 2012. Archived at perma.cc/M7X5-E8SQ

[20] Tom Lianza and Chris Snook. A Byzantine failure in the real world. *blog.cloudflare.com*, November 2020. Archived at perma.cc/83EZ-ALCY

[21] Mohammed Alfatafta, Basil Alkhatib, Ahmed Alquraan, and Samer Al-Kiswany. Toward a Generic Fault Tolerance Technique for Partial Network Partitioning. At *14th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), November 2020.

[22] Marc A. Donges. Re: bnx2 cards Intermittantly Going Offline. Message to Linux *netdev* mailing list, *spinics.net*, September 2012. Archived at perma.cc/TXP6-H8R3

[23] Troy Toman. Inside a CODE RED: Network Edition. *signalvnoise.com*, September 2020. Archived at perma.cc/BET6-FY25

[24] Kyle Kingsbury. Call Me Maybe: Elasticsearch. *aphyr.com*, June 2014. perma.cc/JK47-S89J

[25] Salvatore Sanfilippo. A Few Arguments About Redis Sentinel Properties and Fail Scenarios. *antirez.com*, October 2014. perma.cc/8XEU-CLM8

[26] Nicolas Liochon. CAP: If All You Have Is a Timeout, Everything Looks Like a Partition. *blog.thislongrun.com*, May 2015. Archived at perma.cc/FS57-V2PZ

[27] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. Queues Don't Matter When You Can JUMP Them! At *12th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), May 2015.

[28] Theo Julienne. Debugging network stalls on Kubernetes. *github.blog*, November 2019. Archived at perma.cc/K9M8-XVGL

[29] Guohui Wang and T. S. Eugene Ng. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. At *29th IEEE International Conference on Computer Communications* (INFOCOM), March 2010. doi:10.1109/INFCOM.2010.5461931

[30] Brandon Philips. etcd: Distributed Locking and Service Discovery. At *Strange Loop*, September 2014.

[31] Steve Newman. A Systematic Look at EC2 I/O. *blog.scalyr.com*, October 2012. Archived at perma.cc/FL4R-H2VE

[32] Naohiro Hayashibara, Xavier Défago, Rami Yared, and Takuya Katayama. The φ Accrual Failure Detector. Japan Advanced Institute of Science and Technology, School of Information Science, Technical Report IS-RR-2004-010, May 2004. Archived at perma.cc/NSM2-TRYA

[33] Jeffrey Wang. Phi Accrual Failure Detector. *ternarysearch.blogspot.co.uk*, August 2013. perma.cc/L452-AMLV

[34] Srinivasan Keshav. *An Engineering Approach to Computer Networking: ATM Networks, the Internet, and the Telephone Network*. Addison-Wesley Professional, May 1997. ISBN: 978-0-201-63442-6

[35] Othmar Kyas. *ATM Networks*. International Thomson Publishing, 1995. ISBN: 978-1-850-32128-6

[36] Mellanox Technologies. InfiniBand FAQ, Rev 1.3. *network.nvidia.com*, December 2014. Archived at perma.cc/LQJ4-QZVK

[37] Jose Renato Santos, Yoshio Turner, and G. (John) Janakiraman. End-to-End Congestion Control for InfiniBand. At *22nd Annual Joint Conference of the IEEE Computer and Communications Societies* (INFOCOM), April 2003. Also published by HP Laboratories Palo Alto, Tech Report HPL-2002-359. doi:10.1109/INFCOM.2003.1208949

[38] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. At *ACM Symposium on Cloud Computing* (SOCC), November 2014. doi:10.1145/2670979.2670988

[39] Ulrich Windl, David Dalton, Marc Martinec, and Dale R. Worley. The NTP FAQ and HOWTO. *ntp.org*, November 2006.

[40] John Graham-Cumming. How and why the leap second affected Cloudflare DNS. *blog.cloudflare.com*, January 2017. Archived at archive.org

[41] David Holmes. Inside the Hotspot VM: Clocks, Timers and Scheduling Events – Part I – Windows. *blogs.oracle.com*, October 2006. Archived at archive.org

[42] Joran Dirk Greef. Three Clocks are Better than One. *tigerbeetle.com*, August 2021. Archived at perma.cc/5RXG-EU6B

[43] Oliver Yang. Pitfalls of TSC usage. *oliveryang.net*, September 2015. Archived at perma.cc/Z2QY-5FRA

[44] Steve Loughran. Time on Multi-Core, Multi-Socket Servers. *steveloughran.blogspot.co.uk*, September 2015. Archived at perma.cc/7M4S-D4U6

[45] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Dale Woodford, Yasushi Saito, Christopher Taylor, Michal Szymaniak, and Ruth Wang. Spanner: Google's Globally-Distributed Database. At *10th USENIX Symposium on Operating System Design and Implementation* (OSDI), October 2012.

[46] M. Caporaloni and R. Ambrosini. How Closely Can a Personal Computer Clock Track the UTC Timescale Via the Internet? *European Journal of Physics*, volume 23, issue 4, pages L17–L21, June 2012. doi:10.1088/0143-0807/23/4/103

[47] Nelson Minar. A Survey of the NTP Network. *alumni.media.mit.edu*, December 1999. Archived at perma.cc/EV76-7ZV3

[48] Viliam Holub. Synchronizing Clocks in a Cassandra Cluster Pt. 1 – The Problem. *blog.rapid7.com*, March 2014. Archived at perma.cc/N3RV-5LNL

[49] Poul-Henning Kamp. The One-Second War (What Time Will You Die?) *ACM Queue*, volume 9, issue 4, pages 44–48, April 2011. doi:10.1145/1966989.1967009

[50] Nelson Minar. Leap Second Crashes Half the Internet. *somebits.com*, July 2012. Archived at perma.cc/2WB8-D6EU

[51] Christopher Pascoe. Time, Technology and Leaping Seconds. *googleblog.blogspot.co.uk*, September 2011. Archived at perma.cc/U2JL-7E74

[52] Mingxue Zhao and Jeff Barr. Look Before You Leap – The Coming Leap Second and AWS. *aws.amazon.com*, May 2015. Archived at perma.cc/KPE9-XMFM

[53] Darryl Veitch and Kanthaiah Vijayalayan. Network Timing and the 2015 Leap Second. At *17th International Conference on Passive and Active Measurement* (PAM), April

2016. doi:10.1007/978-3-319-30505-9_29

[54] VMware, Inc. Timekeeping in VMware Virtual Machines. *vmware.com*, October 2008. Archived at perma.cc/HM5R-T5NF

[55] Victor Yodaiken. Clock Synchronization in Finance and Beyond. *yodaiken.com*, November 2017. Archived at perma.cc/9XZD-8ZZN

[56] Mustafa Emre Acer, Emily Stark, Adrienne Porter Felt, Sascha Fahl, Radhika Bhargava, Bhanu Dev, Matt Braithwaite, Ryan Sleevi, and Parisa Tabriz. Where the Wild Warnings Are: Root Causes of Chrome HTTPS Certificate Errors. At *ACM SIGSAC Conference on Computer and Communications Security* (CCS), pages 1407–1420, October 2017. doi:10.1145/3133956.3134007

[57] European Securities and Markets Authority. MiFID II / MiFIR: Regulatory Technical and Implementing Standards – Annex I. *esma.europa.eu*, Report ESMA/2015/1464, September 2015. Archived at perma.cc/ZLX9-FGQ3

[58] Luke Bigum. Solving MiFID II Clock Synchronisation With Minimum Spend (Part 1). *catach.blogspot.com*, November 2015. Archived at perma.cc/4J5W-FNM4

[59] Oleg Obleukhov and Ahmad Byagowi. How Precision Time Protocol is being deployed at Meta. *engineering.fb.com*, November 2022. Archived at perma.cc/29G6-UJNW

[60] John Wiseman. gpsjam.org, July 2022.

[61] Josh Levinson, Julien Ridoux, and Chris Munns. It's About Time: Microsecond-Accurate Clocks on Amazon EC2 Instances. *aws.amazon.com*, November 2023. Archived at perma.cc/56M6-5VMZ

[62] Kyle Kingsbury. Call Me Maybe: Cassandra. *aphyr.com*, September 2013. Archived at perma.cc/4MBR-J96V

[63] John Daily. Clocks Are Bad, or, Welcome to the Wonderful World of Distributed Systems. *riak.com*, November 2013. Archived at perma.cc/4XB5-UCXY

[64] Marc Brooker. It's About Time! *brooker.co.za*, November 2023. Archived at perma.cc/N6YK-DRPA

[65] Kyle Kingsbury. The Trouble with Timestamps. *aphyr.com*, October 2013. Archived at perma.cc/W3AM-5VAV

[66] Leslie Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, volume 21, issue 7, pages 558–565, July 1978.

doi:10.1145/359545.359563

[67] Justin Sheehy. There Is No Now: Problems With Simultaneity in Distributed Systems. *ACM Queue*, volume 13, issue 3, pages 36–41, March 2015. doi:10.1145/2733108

[68] Murat Demirbas. Spanner: Google's Globally-Distributed Database. *muratbuffalo.blogspot.co.uk*, July 2013. Archived at perma.cc/6VWR-C9WB

[69] Dahlia Malkhi and Jean-Philippe Martin. Spanner's Concurrency Control. *ACM SIGACT News*, volume 44, issue 3, pages 73–77, September 2013. doi:10.1145/2527748.2527767

[70] Franck Pachot. Achieving Precise Clock Synchronization on AWS. *yugabyte.com*, December 2024. Archived at perma.cc/UYM6-RNBS

[71] Spencer Kimball. Living Without Atomic Clocks: Where CockroachDB and Spanner diverge. *cockroachlabs.com*, January 2022. Archived at perma.cc/AWZ7-RXFT

[72] Murat Demirbas. Use of Time in Distributed Databases (part 4): Synchronized clocks in production databases. *muratbuffalo.blogspot.com*, January 2025. Archived at perma.cc/9WNX-Q9U3

[73] Cary G. Gray and David R. Cheriton. Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency. At *12th ACM Symposium on Operating Systems Principles* (SOSP), December 1989. doi:10.1145/74850.74870

[74] Daniel Sturman, Scott Delap, Max Ross, et al. Roblox Return to Service. *corp.roblox.com*, January 2022. Archived at perma.cc/8ALT-WAS4

[75] Todd Lipcon. Avoiding Full GCs with MemStore-Local Allocation Buffers. *slideshare.net*, February 2011. Archived at https://perma.cc/CH62-2EWJ

[76] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. At *2nd USENIX Symposium on Symposium on Networked Systems Design & Implementation* (NSDI), May 2005.

[77] Mike Shaver. fsyncers and Curveballs. *shaver.off.net*, May 2008. Archived at archive.org

[78] Zhenyun Zhuang and Cuong Tran. Eliminating Large JVM GC Pauses Caused by Background IO Traffic. *engineering.linkedin.com*, February 2016. Archived at perma.cc/ML2M-X9XT

[79] Martin Thompson. Java Garbage Collection Distilled. *mechanical-sympathy.blogspot.co.uk*, July 2013. Archived at perma.cc/DJT3-NQLQ

[80] David Terei and Amit Levy. Blade: A Data Center Garbage Collector. arXiv:1504.02578, April 2015.

[81] Martin Maas, Tim Harris, Krste Asanović, and John Kubiatowicz. Trash Day: Coordinating Garbage Collection in Distributed Systems. At *15th USENIX Workshop on Hot Topics in Operating Systems* (HotOS), May 2015.

[82] Martin Fowler. The LMAX Architecture. *martinfowler.com*, July 2011. Archived at perma.cc/5AV4-N6RJ

[83] Joseph Y. Halpern and Yoram Moses. Knowledge and common knowledge in a distributed environment. *Journal of the ACM* (JACM), volume 37, issue 3, pages 549–587, July 1990. doi:10.1145/79147.79161

[84] Chuzhe Tang, Zhaoguo Wang, Xiaodong Zhang, Qianmian Yu, Binyu Zang, Haibing Guan, and Haibo Chen. Ad Hoc Transactions in Web Applications: The Good, the Bad, and the Ugly. At *ACM International Conference on Management of Data* (SIGMOD), June 2022. doi:10.1145/3514221.3526120

[85] Flavio P. Junqueira and Benjamin Reed. *ZooKeeper: Distributed Process Coordination*. O'Reilly Media, 2013. ISBN: 978-1-449-36130-3

[86] Enis Söztutar. HBase and HDFS: Understanding Filesystem Usage in HBase. At *HBaseCon*, June 2013. Archived at perma.cc/4DXR-9P88

[87] SUSE LLC. SUSE Linux Enterprise High Availability 15 SP6 Administration Guide, Section 12: Fencing and STONITH. *documentation.suse.com*, March 2025. Archived at perma.cc/8LAR-EL9D

[88] Mike Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. At *7th USENIX Symposium on Operating System Design and Implementation* (OSDI), November 2006.

[89] Kyle Kingsbury. etcd 3.4.3. *jepsen.io*, January 2020. Archived at perma.cc/2P3Y-MPWU

[90] Ensar Basri Kahveci. Distributed Locks are Dead; Long Live Distributed Locks! *hazelcast.com*, April 2019. Archived at perma.cc/7FS5-LDXE

[91] Martin Kleppmann. How to do distributed locking. *martin.kleppmann.com*, February 2016. Archived at perma.cc/Y24W-YQ5L

[92] Salvatore Sanfilippo. Is Redlock safe? *antirez.com*, February 2016. Archived at perma.cc/B6GA-9Q6A

[93] Gunnar Morling. Leader Election With S3 Conditional Writes. *www.morling.dev*, August 2024. Archived at perma.cc/7V2N-J78Y

[94] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems* (TOPLAS), volume 4, issue 3, pages 382–401, July 1982. doi:10.1145/357172.357176

[95] Jim N. Gray. Notes on Data Base Operating Systems. in *Operating Systems: An Advanced Course*, Lecture Notes in Computer Science, volume 60, edited by R. Bayer, R. M. Graham, and G. Seegmüller, pages 393–481, Springer-Verlag, 1978. ISBN: 978-3-540-08755-7. Archived at perma.cc/7S9M-2LZU

[96] Brian Palmer. How Complicated Was the Byzantine Empire? *slate.com*, October 2011. Archived at perma.cc/AN7X-FL3N

[97] Leslie Lamport. My Writings. *lamport.azurewebsites.net*, December 2014. Archived at perma.cc/5NNM-SQGR

[98] John Rushby. Bus Architectures for Safety-Critical Embedded Systems. At *1st International Workshop on Embedded Software* (EMSOFT), October 2001. doi:10.1007/3-540-45449-7_22

[99] Jake Edge. ELC: SpaceX Lessons Learned. *lwn.net*, March 2013. Archived at perma.cc/AYX8-QP5X

[100] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. SoK: Consensus in the Age of Blockchains. At *1st ACM Conference on Advances in Financial Technologies* (AFT), October 2019. doi:10.1145/3318041.3355458

[101] Ezra Feilden, Adi Oltean, and Philip Johnston. Why we should train AI in space. White Paper, *starcloud.com*, September 2024. Archived at perma.cc/7Y3S-8UB6

[102] James Mickens. The Saddest Moment. *USENIX ;login*, May 2013. Archived at perma.cc/T7BZ-XCFR

[103] Martin Kleppmann and Heidi Howard. Byzantine Eventual Consistency and the Fundamental Limits of Peer-to-Peer Databases. *arxiv.org*, December 2020. doi:10.48550/arXiv.2012.00472

[104] Martin Kleppmann. Making CRDTs Byzantine Fault Tolerant. At *9th Workshop on Principles and Practice of Consistency for Distributed Data* (PaPoC), April 2022. doi:10.1145/3517209.3524042

[105] Evan Gilman. The Discovery of Apache ZooKeeper's Poison Packet. *pagerduty.com*, May 2015. Archived at perma.cc/RV6L-Y5CQ

[106] Jonathan Stone and Craig Partridge. When the CRC and TCP Checksum Disagree. At *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication* (SIGCOMM), August 2000. doi:10.1145/347059.347561

[107] Evan Jones. How Both TCP and Ethernet Checksums Fail. *evanjones.ca*, October 2015. Archived at perma.cc/9T5V-B8X5

[108] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, volume 35, issue 2, pages 288–323, April 1988. doi:10.1145/42282.42283

[109] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Transactions on Computer Systems* (TOCS), volume 1, issue 3, pages 222–238, August 1983. doi:10.1145/357369.357371

[110] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S. Gunawi. Limplock: Understanding the Impact of Limpware on Scale-out Cloud Systems. At *4th ACM Symposium on Cloud Computing* (SoCC), October 2013. doi:10.1145/2523616.2523627

[111] Josh Snyder and Joseph Lynch. Garbage collecting unhealthy JVMs, a proactive approach. Netflix Technology Blog, *netflixtechblog.medium.com*, November 2019. Archived at perma.cc/8BTA-N3YB

[112] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-Slow at Scale: Evidence of Hardware Performance Faults in Large Production Systems. At *16th USENIX Conference on File and Storage Technologies*, February 2018.

[113] Peng Huang, Chuanxiong Guo, Lidong Zhou, Jacob R. Lorch, Yingnong Dang, Murali Chintalapati, and Randolph Yao. Gray Failure: The Achilles' Heel of Cloud-Scale Systems. At *16th Workshop on Hot Topics in Operating Systems* (HotOS), May 2017. doi:10.1145/3102980.3103005

[114] Chang Lou, Peng Huang, and Scott Smith. Understanding, Detecting and Localizing Partial Failures in Large System Software. At *17th USENIX Symposium on Networked Systems Design and Implementation* (NSDI), February 2020.

[115] Peter Bailis and Ali Ghodsi. Eventual Consistency Today: Limitations, Extensions, and Beyond. *ACM Queue*, volume 11, issue 3, pages 55-63, March 2013. doi:10.1145/2460276.2462076

[116] Bowen Alpern and Fred B. Schneider. Defining Liveness. *Information Processing Letters*, volume 21, issue 4, pages 181–185, October 1985. doi:10.1016/0020-0190(85)90056-0

[117] Flavio P. Junqueira. Dude, Where's My Metadata? *fpj.me*, May 2015. Archived at perma.cc/D2EU-Y9S5

[118] Scott Sanders. January 28th Incident Report. *github.com*, February 2016. Archived at perma.cc/5GZR-88TV

[119] Jay Kreps. A Few Notes on Kafka and Jepsen. *blog.empathybox.com*, September 2013. perma.cc/XJ5C-F583

[120] Marc Brooker and Ankush Desai. Systems Correctness Practices at AWS. *Queue, Volume 22, Issue 6*, November/December 2024. doi:10.1145/3712057

[121] Andrey Satarin. Testing Distributed Systems: Curated list of resources on testing distributed systems. *asatarin.github.io*. Archived at perma.cc/U5V8-XP24

[122] Phil Eaton and Joran Dirk Greef. We Put a Distributed Database In the Browser – And Made a Game of It!. *tigerbeetle.com*, June 2023. Archived at L7M7-X4HD

[123] Apple, Inc and the FoundationDB project authors. FoundationDB - Simulation and Testing. *apple.github.io*. Archived at perma.cc/4C4L-AUH3

[124] Jack Vanlightly. Verifying Kafka transactions - Diary entry 2 - Writing an initial TLA+ spec. *jack-vanlightly.com*, December 2024. Archived at perma.cc/NSQ8-MQ5N

[125] Siddon Tang. From Chaos to Order — Tools and Techniques for Testing TiDB, A Distributed NewSQL Database. *pingcap.com*, April 2018. Archived at perma.cc/5EJB-R29F

[126] Nathan VanBenschoten. Parallel Commits: An atomic commit protocol for globally distributed transactions. *cockroachlabs.com*, November 2019. Archived at perma.cc/5FZ7-QK6J

[127] Jack Vanlightly. Paper: VR Revisited - State Transfer (part 3). *jack-vanlightly.com*, December 2022. Archived at perma.cc/KNK3-K6WS

[128] Hillel Wayne. What if the spec doesn't match the code? *buttondown.com*, March 2024. Archived at perma.cc/8HEZ-KHER

[129] Lingzhi Ouyang, Xudong Sun, Ruize Tang, Yu Huang, Madhav Jivrajani, Xiaoxing Ma, Tianyin Xu. Multi-Grained Specifications for Distributed System Model Checking and Verification. At *20th European Conference on Computer Systems* (EuroSys), March 2025. doi:10.1145/3689031.3696069

[130] Yury Izrailevsky and Ariel Tseitlin. The Netflix Simian Army. *netflixtechblog.com*, July, 2011. Archived at perma.cc/M3NY-FJW6

[131] Kyle Kingsbury. Jepsen: On the perils of network partitions. *aphyr.com*, May, 2013. Archived at perma.cc/W98G-6HQP

[132] Kyle Kingsbury. Jepsen Analyses. *jepsen.io*, 2024. Archived at perma.cc/8LDN-D2T8

[133] Rupak Majumdar and Filip Niksic. Why is random testing effective for partition tolerance bugs? *Proceedings of the ACM on Programming Languages* (PACMPL), volume 2, issue POPL, article no. 46, December 2017. doi:10.1145/3158134

[134] FoundationDB project authors. Simulation and Testing. *apple.github.io*. Archived at perma.cc/NQ3L-PM4C

[135] Alex Kladov. Simulation Testing For Liveness. *tigerbeetle.com*, July 2023. Archived at perma.cc/RKD4-HGCR

[136] Alfonso Subiotto Marqués. (Mostly) Deterministic Simulation Testing in Go. *polarsignals.com*, May 2024. Archived at perma.cc/ULD6-TSA4