PART II
Distributed Systems

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

Without distributed systems, we wouldn't be able to make phone calls, transfer money, or exchange information over long distances. We use distributed systems daily. Sometimes, even without acknowledging it: any client/server application is a distributed system.

For many modern software systems, *vertical* scaling (scaling by running the same software on a bigger, faster machine with more CPU, RAM, or faster disks) isn't viable. Bigger machines are more expensive, harder to replace, and may require special maintenance. An alternative is to scale *horizontally*: to run software on multiple machines connected over the network and working as a single logical entity.

Distributed systems might differ both in size, from a handful to hundreds of machines, and in characteristics of their participants, from small handheld or sensor devices to high-performance computers.

The time when database systems were mainly running on a single node is long gone, and most modern database systems have multiple nodes connected in clusters to increase storage capacity, improve performance, and enhance availability.

Even though some of the theoretical breakthroughs in distributed computing aren't new, most of their practical application happened relatively recently. Today, we see increasing interest in the subject, more research, and new development being done.

Basic definitions

In a distributed system, we have several *participants* (sometimes called *processes*, *nodes*, or *replicas*). Each participant has its own local *state*. Participants communicate by exchanging *messages* using communication *links* between them.

Processes can access the time using a *clock*, which can be *logical* or *physical*. Logical clocks are implemented using a kind of monotonically growing counter. Physical clocks, also called *wall clocks*, are bound to a notion of time in the physical world and are accessible through process-local means; for example, through an operating system.

It's impossible to talk about distributed systems without mentioning the inherent difficulties caused by the fact that its parts are located apart from each other. Remote processes communicate through links that can be slow and unreliable, which makes knowing the exact state of the remote process more complicated.

Most of the research in the distributed systems field is related to the fact that nothing is entirely reliable: communication channels may delay, reorder, or fail to deliver the messages; processes may pause, slow down, crash, go out of control, or suddenly stop responding.

There are many themes in common in the fields of concurrent and distributed programming, since CPUs are tiny distributed systems with links, processors, and communication protocols. You'll see many parallels with concurrent programming in "Consistency Models" on page 222. However, most of the primitives can't be reused directly because of the costs of communication between remote parties, and the unreliability of links and processes.

To overcome the difficulties of the distributed environment, we need to use a particular class of algorithms, *distributed algorithms*, which have notions of local and remote state and execution and work despite unreliable networks and component failures. We describe algorithms in terms of *state* and *steps* (or *phases*), with *transitions* between them. Each process executes the algorithm steps locally, and a combination of local executions and process interactions constitutes a distributed algorithm.

Distributed algorithms describe the local behavior and interaction of multiple independent nodes. Nodes communicate by sending messages to each other. Algorithms define participant roles, exchanged messages, states, transitions, executed steps, properties of the delivery medium, timing assumptions, failure models, and other characteristics that describe processes and their interactions.

Distributed algorithms serve many different purposes:

Coordination
: A process that supervises the actions and behavior of several workers.

Cooperation
: Multiple participants relying on one another for finishing their tasks.

Dissemination
: Processes cooperating in spreading the information to all interested parties quickly and reliably.

Consensus
: Achieving agreement among multiple processes.

In this book, we talk about algorithms in the context of their usage and prefer a practical approach over purely academic material. First, we cover all necessary abstractions, the processes and the connections between them, and progress to building more complex communication patterns. We start with UDP, where the sender doesn't have any guarantees on whether or not its message has reached its destination; and finally, to achieve consensus, where multiple processes agree on a specific value.

CHAPTER 8
Introduction and Overview

What makes distributed systems inherently different from single-node systems? Let's take a look at a simple example and try to see. In a single-threaded program, we define variables and the execution process (a set of steps).

For example, we can define a variable and perform simple arithmetic operations over it:

```
int i = 1;
i += 2;
i *= 2;
```

We have a single execution history: we declare a variable, increment it by two, then multiply it by two, and get the result: `6`. Let's say that, instead of having one execution thread performing these operations, we have two threads that have read and write access to variable `x`.

Concurrent Execution

As soon as two execution threads are allowed to access the variable, the exact outcome of the concurrent step execution is unpredictable, unless the steps are synchronized between the threads. Instead of a single possible outcome, we end up with four, as Figure 8-1 shows.[1]

1 Interleaving, where the multiplier reads before the adder, is left out for brevity, since it yields the same result as a).

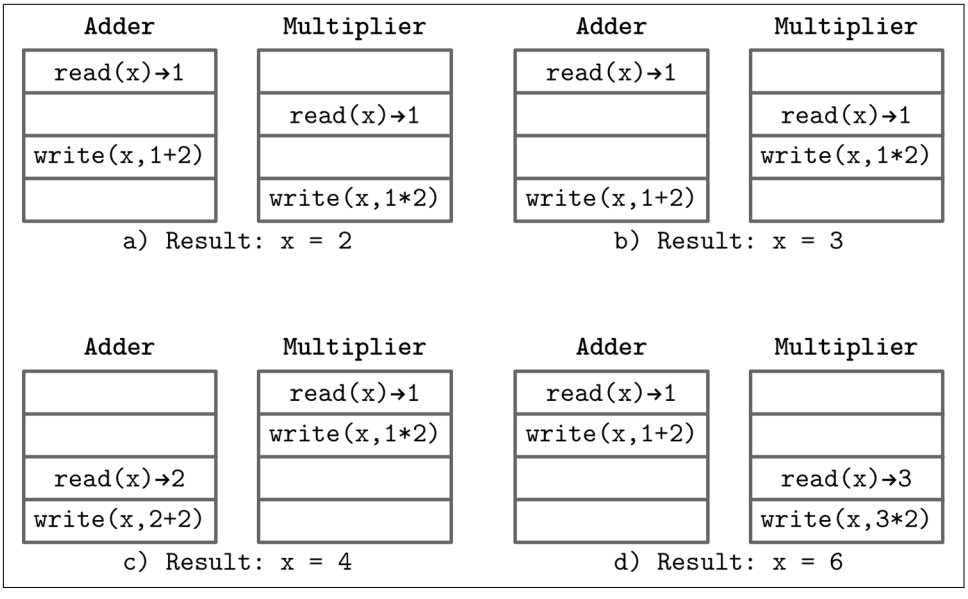

Figure 8-1. Possible interleavings of concurrent executions

- a) x = 2, if both threads read an initial value, the adder writes its value, but it is overwritten with the multiplication result.
- b) x = 3, if both threads read an initial value, the multiplier writes its value, but it is overwritten with the addition result.
- c) x = 4, if the multiplier can read the initial value and execute its operation before the adder starts.
- d) x = 6, if the adder can read the initial value and execute its operation before the multiplier starts.

Even before we can cross a single node boundary, we encounter the first problem in distributed systems: *concurrency*. Every concurrent program has some properties of a distributed system. Threads access the shared state, perform some operations locally, and propagate the results back to the shared variables.

To define execution histories precisely and reduce the number of possible outcomes, we need *consistency models*. Consistency models describe concurrent executions and establish an order in which operations can be executed and made visible to the participants. Using different consistency models, we can constraint or relax the number of states the system can be in.

There is a lot of overlap in terminology and research in the areas of distributed systems and concurrent computing, but there are also some differences. In a concurrent system, we can have *shared memory*, which processors can use to exchange the

information. In a distributed system, each processor has its local state and participants communicate by passing messages.

Concurrent and Parallel

We often use the terms *concurrent* and *parallel* computing interchangeably, but these concepts have a slight semantic difference. When two sequences of steps execute concurrently, both of them are in progress, but only one of them is executed at any moment. If two sequences execute in parallel, their steps can be executed simultaneously. Concurrent operations overlap in time, while parallel operations are executed by multiple processors [WEIKUM01].

Joe Armstrong, creator of the Erlang programming language, gave an example: concurrent execution is like having two queues to a single coffee machine, while parallel execution is like having two queues to two coffee machines. That said, the vast majority of sources use the term concurrency to describe systems with several parallel execution threads, and the term parallelism is rarely used.

Shared State in a Distributed System

We can try to introduce some notion of shared memory to a distributed system, for example, a single source of information, such as database. Even if we solve the problems with concurrent access to it, we still cannot guarantee that all processes are in sync.

To access this database, processes have to go over the communication medium by sending and receiving messages to query or modify the state. However, what happens if one of the processes does not receive a response from the database for a longer time? To answer this question, we first have to define what *longer* even means. To do this, the system has to be described in terms of *synchrony*: whether the communication is fully asynchronous, or whether there are some timing assumptions. These timing assumptions allow us to introduce operation timeouts and retries.

We do not know whether the database hasn't responded because it's overloaded, unavailable, or slow, or because of some problems with the network on the way to it. This describes a *nature* of a crash: processes may crash by failing to participate in further algorithm steps, having a temporary failure, or by omitting some of the messages. We need to define a *failure model* and describe ways in which failures can occur before we decide how to treat them.

A property that describes system reliability and whether or not it can continue operating correctly in the presence of failures is called *fault tolerance*. Failures are inevitable, so we need to build systems with reliable components, and eliminating a single point of failure in the form of the aforementioned single-node database can be the

first step in this direction. We can do this by introducing some *redundancy* and adding a backup database. However, now we face a different problem: how do we keep *multiple copies* of shared state in sync?

So far, trying to introduce shared state to our simple system has left us with more questions than answers. We now know that sharing state is not as simple as just introducing a database, and have to take a more granular approach and describe interactions in terms of independent processes and passing messages between them.

Fallacies of Distributed Computing

In an ideal case, when two computers talk over the network, everything works just fine: a process opens up a connection, sends the data, gets responses, and everyone is happy. Assuming that operations always succeed and nothing can go wrong is dangerous, since when something does break and our assumptions turn out to be wrong, systems behave in ways that are hard or impossible to predict.

Most of the time, assuming that the *network is reliable* is a reasonable thing to do. It has to be reliable to at least some extent to be useful. We've all been in the situation when we tried to establish a connection to the remote server and got a `Network is Unreachable` error instead. But even if it is possible to establish a connection, a successful *initial* connection to the server does not guarantee that the link is stable, and the connection can get interrupted at any time. The message might've reached the remote party, but the response could've gotten lost, or the connection was interrupted before the response was delivered.

Network switches break, cables get disconnected, and network configurations can change at any time. We should build our system by handling all of these scenarios gracefully.

A connection can be stable, but we can't expect remote calls to be as fast as the local ones. We should make as few assumptions about latency as possible and never assume that *latency is zero*. For our message to reach a remote server, it has to go through several software layers, and a physical medium such as optic fiber or a cable. All of these operations are not instantaneous.

Michael Lewis, in his *Flash Boys* book (Simon and Schuster), tells a story about companies spending millions of dollars to reduce latency by several milliseconds to able to access stock exchanges faster than the competition. This is a great example of using latency as a competitive advantage, but it's worth mentioning that, according to some other studies, such as [BARTLETT16], the chance of stale-quote arbitrage (the ability to profit from being able to know prices and execute orders faster than the competition) doesn't give fast traders the ability to exploit markets.

Learning our lessons, we've added retries, reconnects, and removed the assumptions about instantaneous execution, but this still turns out not to be enough. When increasing the number, rates, and sizes of exchanged messages, or adding new processes to the existing network, we should not assume that *bandwidth is infinite*.

In 1994, Peter Deutsch published a now-famous list of assertions, titled "Fallacies of distributed computing," describing the aspects of distributed computing that are easy to overlook. In addition to network reliability, latency, and bandwidth assumptions, he describes some other problems. For example, network security, the possible presence of adversarial parties, intentional and unintentional topology changes that can break our assumptions about presence and location of specific resources, transport costs in terms of both time and resources, and, finally, the existence of a single authority having knowledge and control over the entire network.

Deutsch's list of distributed computing fallacies is pretty exhaustive, but it focuses on what can go wrong when we send messages from one process to another through the link. These concerns are valid and describe the most general and low-level complications, but unfortunately, there are many other assumptions we make about the distributed systems while designing and implementing them that can cause problems when operating them.

Processing

Before a remote process can send a response to the message it just received, it needs to perform some work locally, so we cannot assume that *processing is instantaneous*. Taking network latency into consideration is not enough, as operations performed by the remote processes aren't immediate, either.

Moreover, there's no guarantee that processing starts as soon as the message is delivered. The message may land in the pending queue on the remote server, and will have to wait there until all the messages that arrived before it are processed.

Nodes can be located closer or further from one another, have different CPUs, amounts of RAM, different disks, or be running different software versions and configurations. We cannot expect them to process requests at the same rate. If we have to wait for several remote servers working in parallel to respond to complete the task, the execution as a whole is as slow as the slowest remote server.

Contrary to the widespread belief, *queue capacity is not infinite* and piling up more requests won't do the system any good. *Backpressure* is a strategy that allows us to cope with producers that publish messages at a rate that is faster than the rate at which consumers can process them by slowing down the producers. Backpressure is

one of the least appreciated and applied concepts in distributed systems, often built post hoc instead of being an integral part of the system design.

Even though increasing the queue capacity might sound like a good idea and can help to pipeline, parallelize, and effectively schedule requests, nothing is happening to the messages while they're sitting in the queue and waiting for their turn. Increasing the queue size may negatively impact latency, since changing it has no effect on the processing rate.

In general, process-local queues are used to achieve the following goals:

Decoupling
: Receipt and processing are separated in time and happen independently.

Pipelining
: Requests in different stages are processed by independent parts of the system. The subsystem responsible for receiving messages doesn't have to block until the previous message is fully processed.

Absorbing short-time bursts
: System load tends to vary, but request inter-arrival times are hidden from the component responsible for request processing. Overall system latency increases because of the time spent in the queue, but this is usually still better than responding with a failure and retrying the request.

Queue size is workload- and application-specific. For relatively stable workloads, we can size queues by measuring task processing times and the average time each task spends in the queue before it is processed, and making sure that latency remains within acceptable bounds while throughput increases. In this case, queue sizes are relatively small. For unpredictable workloads, when tasks get submitted in bursts, queues should be sized to account for bursts and high load as well.

The remote server can work through requests quickly, but it doesn't mean that we always get a positive response from it. It can respond with a failure: it couldn't make a write, the searched value was not present, or it could've hit a bug. In summary, even the most favorable scenario still requires some attention from our side.

Clocks and Time

> Time is an illusion. Lunchtime doubly so.
>
> —Ford Prefect, *The Hitchhiker's Guide to the Galaxy*

Assuming that clocks on remote machines run in sync can also be dangerous. Combined with *latency is zero* and *processing is instantaneous*, it leads to different idiosyncrasies, especially in time-series and real-time data processing. For example, when collecting and aggregating data from participants with a different perception of time,

you should understand time drifts between them and normalize times accordingly, rather than relying on the source timestamp. Unless you use specialized high-precision time sources, you should not rely on timestamps for synchronization or ordering. Of course this doesn't mean we cannot or should not rely on time at all: in the end, any synchronous system uses *local* clocks for timeouts.

It's essential to always account for the possible time differences between the processes and the time required for the messages to get delivered and processed. For example, Spanner (see "Distributed Transactions with Spanner" on page 268) uses a special time API that returns a timestamp and uncertainty bounds to impose a strict transaction order. Some failure-detection algorithms rely on a shared notion of time and a guarantee that the clock drift is always within allowed bounds for correctness [GUPTA01].

Besides the fact that clock synchronization in a distributed system is hard, the *current* time is constantly changing: you can request a current POSIX timestamp from the operating system, and request another *current* timestamp after executing several steps, and the two will be different. This is a rather obvious observation, but understanding both a source of time and which exact moment the timestamp captures is crucial.

Understanding whether the clock source is monotonic (i.e., that it won't ever go backward) and how much the scheduled time-related operations might drift can be helpful, too.

State Consistency

Most of the previous assumptions fall into the *almost always false* category, but there are some that are better described as *not always true*: when it's easy to take a mental shortcut and simplify the model by thinking of it a specific way, ignoring some tricky edge cases.

Distributed algorithms do not always guarantee strict state consistency. Some approaches have looser constraints and allow state divergence between replicas, and rely on *conflict resolution* (an ability to detect and resolve diverged states within the system) and *read-time data repair* (bringing replicas back in sync during reads in cases where they respond with different results). You can find more information about these concepts in Chapter 12. Assuming that the state is fully consistent across the nodes may lead to subtle bugs.

An eventually consistent distributed database system might have the logic to handle replica disagreement by querying a quorum of nodes during reads, but assume that the database schema and the view of the cluster are strongly consistent. Unless we enforce consistency of this information, relying on that assumption may have severe consequences.

For example, there was a bug in Apache Cassandra, caused by the fact that schema changes propagate to servers at different times. If you tried to read from the database while the schema was propagating, there was a chance of corruption, since one server encoded results assuming one schema and the other one decoded them using a different schema.

Another example is a bug caused by the divergent view of the ring: if one of the nodes assumes that the other node holds data records for a key, but this other node has a different view of the cluster, reading or writing the data can result in misplacing data records or getting an empty response while data records are in fact happily present on the other node.

It is better to think about the possible problems in advance, even if a complete solution is costly to implement. By understanding and handling these cases, you can embed safeguards or change the design in a way that makes the solution more natural.

Local and Remote Execution

Hiding complexity behind an API might be dangerous. For example, if you have an iterator over the local dataset, you can reasonably predict what's going on behind the scenes, even if the storage engine is unfamiliar. Understanding the process of iteration over the remote dataset is an entirely different problem: you need to understand consistency and delivery semantics, data reconciliation, paging, merges, concurrent access implications, and many other things.

Simply hiding both behind the same interface, however useful, might be misleading. Additional API parameters may be necessary for debugging, configuration, and observability. We should always keep in mind that *local and remote execution are not the same* [WALDO96].

The most apparent problem with hiding remote calls is latency: remote invocation is many times more costly than the local one, since it involves two-way network transport, serialization/deserialization, and many other steps. Interleaving local and blocking remote calls may lead to performance degradation and unintended side effects [VINOSKI08].

Need to Handle Failures

It's OK to start working on a system assuming that all nodes are up and functioning normally, but thinking this is the case all the time is dangerous. In a long-running system, nodes can be taken down for maintenance (which usually involves a graceful shutdown) or crash for various reasons: software problems, out-of-memory killer

[KERRISK10], runtime bugs, hardware issues, etc. Processes do fail, and the best thing you can do is be prepared for failures and understand how to handle them.

If the remote server doesn't respond, we do not always know the exact reason for it. It could be caused by the crash, a network failure, the remote process, or the link to it being slow. Some distributed algorithms use *heartbeat protocols* and *failure detectors* to form a hypothesis about which participants are alive and reachable.

Network Partitions and Partial Failures

When two or more servers cannot communicate with each other, we call the situation *network partition*. In "Perspectives on the CAP Theorem" [GILBERT12], Seth Gilbert and Nancy Lynch draw a distinction between the case when two participants cannot communicate with each other and when several groups of participants are isolated from one another, cannot exchange messages, and proceed with the algorithm.

General unreliability of the network (packet loss, retransmission, latencies that are hard to predict) are *annoying but tolerable*, while network partitions can cause much more trouble, since independent groups can proceed with execution and produce conflicting results. Network links can also fail asymmetrically: messages can still be getting delivered from one process to the other one, but not vice versa.

To build a system that is robust in the presence of failure of one or multiple processes, we have to consider cases of *partial failures* [TANENBAUM06] and how the system can continue operating even though a part of it is unavailable or functioning incorrectly.

Failures are hard to detect and aren't always visible in the same way from different parts of the system. When designing highly available systems, one should always think about edge cases: what if we did replicate the data, but received no acknowledgments? Do we need to retry? Is the data still going to be available for reads on the nodes that have sent acknowledgments?

Murphy's Law[2] tells us that the failures do happen. Programming folklore adds that the failures will happen in the worst way possible, so our job as distributed systems engineers is to make sure we reduce the number of scenarios where things go wrong and prepare for failures in a way that contains the damage they can cause.

It's impossible to prevent all failures, but we can still build a resilient system that functions correctly in their presence. The best way to design for failures is to test for them. It's close to impossible to think through every possible failure scenario and predict the behaviors of multiple processes. Setting up testing harnesses that create partitions,

2 Murphy's Law is an adage that can be summarized as "Anything that can go wrong, will go wrong," which was popularized and is often used as an idiom in popular culture.

simulate bit rot [GRAY05], increase latencies, diverge clocks, and magnify relative processing speeds is the best way to go about it. Real-world distributed system setups can be quite adversarial, unfriendly, and "creative" (however, in a very hostile way), so the testing effort should attempt to cover as many scenarios as possible.

Over the last few years, we've seen a few open source projects that help to recreate different failure scenarios. Toxiproxy can help to simulate network problems: limit the bandwidth, introduce latency, timeouts, and more. Chaos Monkey takes a more radical approach and exposes engineers to production failures by randomly shutting down services. CharybdeFS helps to simulate filesystem and hardware errors and failures. You can use these tools to test your software and make sure it behaves correctly in the presence of these failures. CrashMonkey, a filesystem agnostic record-replay-and-test framework, helps test data and metadata consistency for persistent files.

When working with distributed systems, we have to take fault tolerance, resilience, possible failure scenarios, and edge cases seriously. Similar to "given enough eyeballs, all bugs are shallow," we can say that a large enough cluster will eventually hit every possible issue. At the same time, given enough testing, we will be able to eventually find every existing problem.

Cascading Failures

We cannot always wholly isolate failures: a process tipping over under a high load increases the load for the rest of cluster, making it even more probable for the other nodes to fail. *Cascading failures* can propagate from one part of the system to the other, increasing the scope of the problem.

Sometimes, cascading failures can even be initiated by perfectly good intentions. For example, a node was offline for a while and did not receive the most recent updates. After it comes back online, helpful peers would like to help it to catch up with recent happenings and start streaming the data it's missing over to it, exhausting network resources or causing the node to fail shortly after the startup.

To protect a system from propagating failures and treat failure scenarios gracefully, *circuit breakers* can be used. In electrical engineering, circuit breakers protect expensive and hard-to-replace parts from overload or short circuit by interrupting the current flow. In software development, circuit breakers monitor failures and allow fallback mechanisms that can protect the system by steering away from the failing service, giving it some time to recover, and handling failing calls gracefully.

When the connection to one of the servers fails or the server does not respond, the client starts a reconnection loop. By that point, an overloaded server already has a hard time catching up with new connection requests, and client-side retries in a tight loop don't help the situation. To avoid that, we can use a *backoff* strategy. Instead of retrying immediately, clients wait for some time. Backoff can help us to avoid amplifying problems by scheduling retries and increasing the time window between subsequent requests.

Backoff is used to increase time periods between requests from a single client. However, different clients using the same backoff strategy can produce substantial load as well. To prevent *different* clients from retrying all at once after the backoff period, we can introduce *jitter*. Jitter adds small random time periods to backoff and reduces the probability of clients waking up and retrying at the same time.

Hardware failures, bit rot, and software errors can result in corruption that can propagate through standard delivery mechanisms. For example, corrupted data records can get replicated to the other nodes if they are not validated. Without validation mechanisms in place, a system can propagate corrupted data to the other nodes, potentially overwriting noncorrupted data records. To avoid that, we should use checksumming and validation to verify the integrity of any content exchanged between the nodes.

Overload and hotspotting can be avoided by planning and coordinating execution. Instead of letting peers execute operation steps independently, we can use a coordinator that prepares an execution plan based on the available resources and predicts the load based on the past execution data available to it.

In summary, we should always consider cases in which failures in one part of the system can cause problems elsewhere. We should equip our systems with circuit breakers, backoff, validation, and coordination mechanisms. Handling small isolated problems is always more straightforward than trying to recover from a large outage.

We've just spent an entire section discussing problems and potential failure scenarios in distributed systems, but we should see this as a warning and not as something that should scare us away.

Understanding what can go wrong, and carefully designing and testing our systems makes them more robust and resilient. Being aware of these issues can help you to identify and find potential sources of problems during development, as well as debug them in production.

Distributed Systems Abstractions

When talking about programming languages, we use common terminology and define our programs in terms of functions, operators, classes, variables, and pointers.

Having a common vocabulary helps us to avoid inventing new words every time we describe anything. The more precise and less ambiguous our definitions are, the easier it is for our listeners to understand us.

Before we move to algorithms, we first have to cover the distributed systems vocabulary: definitions you'll frequently encounter in talks, books, and papers.

Links

Networks are not reliable: messages can get lost, delayed, and reordered. Now, with this thought in our minds, we will try to build several communication protocols. We'll start with the least reliable and robust ones, identifying the states they can be in, and figuring out the possible additions to the protocol that can provide better guarantees.

Fair-loss link

We can start with two *processes*, connected with a *link*. Processes can send messages to each other, as shown in Figure 8-2. Any communication medium is imperfect, and messages can get lost or delayed.

Let's see what kind of guarantees we can get. After the message `M` is sent, from the senders' perspective, it can be in one of the following states:

- Not *yet* delivered to process `B` (but will be, at some point in time)
- Irrecoverably lost during transport
- Successfully delivered to the remote process

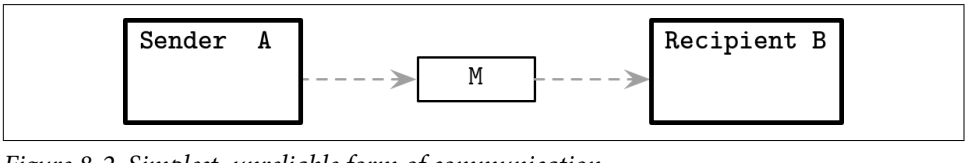

Figure 8-2. Simplest, unreliable form of communication

Notice that the sender does not have any way to find out if the message is already delivered. In distributed systems terminology, this kind of link is called *fair-loss*. The properties of this kind of link are:

Fair loss
: If both sender and recipient are correct and the sender keeps retransmitting the message infinitely many times, it will eventually be delivered.[3]

Finite duplication
: Sent messages won't be delivered infinitely many times.

No creation
: A link will not come up with messages; in other words, it won't deliver the message that was never sent.

A fair-loss link is a useful abstraction and a first building block for communication protocols with strong guarantees. We can assume that this link is not losing messages between communicating parties *systematically* and doesn't create new messages. But, at the same time, we cannot entirely rely on it. This might remind you of the User Datagram Protocol (UDP), which allows us to send messages from one process to the other, but does not have reliable delivery semantics on the protocol level.

Message acknowledgments

To improve the situation and get more clarity in terms of message status, we can introduce *acknowledgments*: a way for the recipient to notify the sender that it has received the message. For that, we need to use bidirectional communication channels and add some means that allow us to distinguish differences between the messages; for example, *sequence numbers*, which are unique monotonically increasing message identifiers.

It is enough to have a *unique* identifier for every message. Sequence numbers are just a particular case of a unique identifier, where we achieve uniqueness by drawing identifiers from a counter. When using hash algorithms to identify messages uniquely, we should account for possible collisions and make sure we can still disambiguate messages.

Now, process `A` can send a message `M(n)`, where `n` is a monotonically increasing message counter. As soon as `B` receives the message, it sends an acknowledgment `ACK(n)` back to A. Figure 8-3 shows this form of communication.

3 A more precise definition is that if a correct process A sends a message to a correct process B infinitely often, it will be delivered infinitely often ([CACHIN11]).

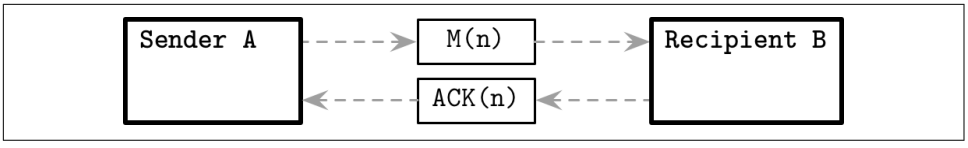

Figure 8-3. Sending a message with an acknowledgment

The acknowledgment, as well as the original message, may get lost on the way. The number of states the message can be in changes slightly. Until `A` receives an acknowledgment, the message is still in one of the three states we mentioned previously, but as soon as `A` receives the acknowledgment, it can be confident that the message is delivered to `B`.

Message retransmits

Adding acknowledgments is *still* not enough to call this communication protocol reliable: a sent message may still get lost, or the remote process may fail before acknowledging it. To solve this problem and provide delivery guarantees, we can try *retransmits* instead. Retransmits are a way for the sender to retry a potentially failed operation. We say *potentially* failed, because the sender doesn't really know whether it has failed or not, since the type of link we're about to discuss does *not* use acknowledgments.

After process `A` sends message `M`, it waits until timeout `T` is triggered and tries to send the same message again. Assuming the link between processes stays intact, network partitions between the processes are not infinite, and not *all* packets are lost, we can state that, from the sender's perspective, the message is either not *yet* delivered to process `B` or is successfully delivered to process `B`. Since `A` keeps trying to send the message, we can say that it *cannot* get irrecoverably lost during transport.

In distributed systems terminology, this abstraction is called a *stubborn link*. It's called stubborn because the sender keeps resending the message again and again indefinitely, but, since this sort of abstraction would be highly impractical, we need to combine retries with acknowledgments.

Problem with retransmits

Whenever we send the message, until we receive an acknowledgment from the remote process, we do not know whether it has already been processed, it will be processed shortly, it has been lost, or the remote process has crashed before receiving it—any one of these states is possible. We can retry the operation and send the message again, but this can result in message duplicates. Processing duplicates is only safe if the operation we're about to perform is idempotent.

An *idempotent* operation is one that can be executed multiple times, yielding the same result without producing additional side effects. For example, a server shut-

down operation can be idempotent, the first call initiates the shutdown, and all subsequent calls do not produce any additional effects.

If every operation was idempotent, we could think less about delivery semantics, rely more on retransmits for fault tolerance, and build systems in an entirely reactive way: triggering an action as a response to some signal, without causing unintended side effects. However, operations are not necessarily idempotent, and merely assuming that they are might lead to cluster-wide side effects. For example, charging a customer's credit card is not idempotent, and charging it multiple times is definitely undesirable.

Idempotence is particularly important in the presence of partial failures and network partitions, since we cannot always find out the exact status of a remote operation—whether it has succeeded, failed, or will be executed shortly—and we just have to wait longer. Since guaranteeing that each executed operation is idempotent is an unrealistic requirement, we need to provide guarantees *equivalent* to idempotence without changing the underlying operation semantics. To achieve this, we can use *deduplication* and avoid processing messages more than once.

Message order

Unreliable networks present us with two problems: messages can arrive out of order and, because of retransmits, some messages may arrive more than once. We have already introduced sequence numbers, and we can use these message identifiers on the recipient side to ensure *first-in, first-out* (FIFO) ordering. Since every message has a sequence number, the receiver can track:

- $n_{consecutive}$, specifying the highest sequence number, up to which it has seen all messages. Messages up to this number can be put back in order.
- $n_{processed}$, specifying the highest sequence number, up to which messages were put back in their original order and *processed*. This number can be used for deduplication.

If the received message has a nonconsecutive sequence number, the receiver puts it into the reordering buffer. For example, it receives a message with a sequence number `5` after receiving one with `3`, and we know that `4` is still missing, so we need to put `5` aside until `4` comes, and we can reconstruct the message order. Since we're building on top of a fair-loss link, we assume that messages between $n_{consecutive}$ and n_{max_seen} will eventually be delivered.

The recipient can safely discard the messages with sequence numbers up to $n_{consecutive}$ that it receives, since they're guaranteed to be already delivered.

Deduplication works by checking if the message with a sequence number `n` has already been *processed* (passed down the stack by the receiver) and discarding already processed messages.

In distributed systems terms, this type of link is called a *perfect link*, which provides the following guarantees [CACHIN11]:

Reliable delivery
: Every message sent *once* by the correct process `A` to the correct process `B`, will *eventually* be delivered.

No duplication
: No message is delivered more than once.

No creation
: Same as with other types of links, it can only deliver the messages that were actually sent.

This might remind you of the TCP[4] protocol (however, reliable delivery in TCP is guaranteed only in the scope of a single session). Of course, this model is just a simplified representation we use for illustration purposes only. TCP has a much more sophisticated model for dealing with acknowledgments, which groups acknowledgments and reduces the protocol-level overhead. In addition, TCP has selective acknowledgments, flow control, congestion control, error detection, and many other features that are out of the scope of our discussion.

Exactly-once delivery

> There are only two hard problems in distributed systems: 2. Exactly-once delivery 1. Guaranteed order of messages 2. Exactly-once delivery.
>
> —Mathias Verraes

There have been many discussions about whether or not *exactly-once delivery* is possible. Here, semantics and precise wording are essential. Since there might be a link failure preventing the message from being delivered from the first try, most of the real-world systems employ *at-least-once delivery*, which ensures that the sender retries until it receives an acknowledgment, otherwise the message is not considered to be received. Another delivery semantic is *at-most-once*: the sender sends the message and doesn't expect any delivery confirmation.

The TCP protocol works by breaking down messages into packets, transmitting them one by one, and stitching them back together on the receiving side. TCP might attempt to retransmit some of the packets, and more than one transmission attempt

4 See *https://databass.dev/links/53*.

may succeed. Since TCP marks each packet with a sequence number, even though some packets were transmitted more than once, it can deduplicate the packets and guarantee that the recipient will see the message and *process* it only once. In TCP, this guarantee is valid only for a *single session*: if the message is acknowledged and processed, but the sender didn't receive the acknowledgment before the connection was interrupted, the application is not aware of this delivery and, depending on its logic, it might attempt to send the message once again.

This means that exactly-once *processing* is what's interesting here since duplicate *deliveries* (or packet transmissions) have no side effects and are merely an artifact of the best effort by the link. For example, if the database node has only *received* the record, but hasn't *persisted* it, delivery has occurred, but it'll be of no use unless the record can be retrieved (in other words, unless it was both delivered and processed).

For the exactly-once guarantee to hold, nodes should have a *common knowledge* [HALPERN90]: everyone knows about some fact, and everyone knows that everyone else also knows about that fact. In simplified terms, nodes have to agree on the state of the record: both nodes agree that it either *was* or *was not* persisted. As you will see later in this chapter, this is theoretically impossible, but in practice we still use this notion by relaxing coordination requirements.

Any misunderstanding about whether or not exactly-once delivery is possible most likely comes from approaching the problem from different protocol and abstraction levels and the definition of "delivery." It's not possible to build a reliable link without ever transferring any message more than once, but we can create the illusion of exactly-once delivery from the sender's perspective by *processing* the message once and ignoring duplicates.

Now, as we have established the means for reliable communication, we can move ahead and look for ways to achieve uniformity and agreement between processes in the distributed system.

Two Generals' Problem

One of the most prominent descriptions of an agreement in a distributed system is a thought experiment widely known as the *Two Generals' Problem*.

This thought experiment shows that it is impossible to achieve an agreement between two parties if communication is *asynchronous* in the presence of link failures. Even though TCP exhibits properties of a perfect link, it's important to remember that perfect links, despite the name, do not guarantee *perfect* delivery. They also can't guarantee that participants will be alive the whole time, and are concerned only with transport.

Imagine two armies, led by two generals, preparing to attack a fortified city. The armies are located on two sides of the city and can succeed in their siege only if they attack simultaneously.

The generals can communicate by sending messengers, and already have devised an attack plan. The only thing they now have to agree on is whether or not to carry out the plan. Variants of this problem are when one of the generals has a higher rank, but needs to make sure the attack is coordinated; or that the generals need to agree on the exact time. These details do not change the problem definition: the generals have to come to an agreement.

The army generals only have to agree on the fact that they both will proceed with the attack. Otherwise, the attack cannot succeed. General `A` sends a message `MSG(N)`, stating an intention to proceed with the attack at a specified time, *if* the other party agrees to proceed as well.

After `A` sends the messenger, he doesn't know whether the messenger has arrived or not: the messenger can get captured and fail to deliver the message. When general `B` receives the message, he has to send an acknowledgment `ACK(MSG(N))`. Figure 8-4 shows that a message is sent one way and acknowledged by the other party.

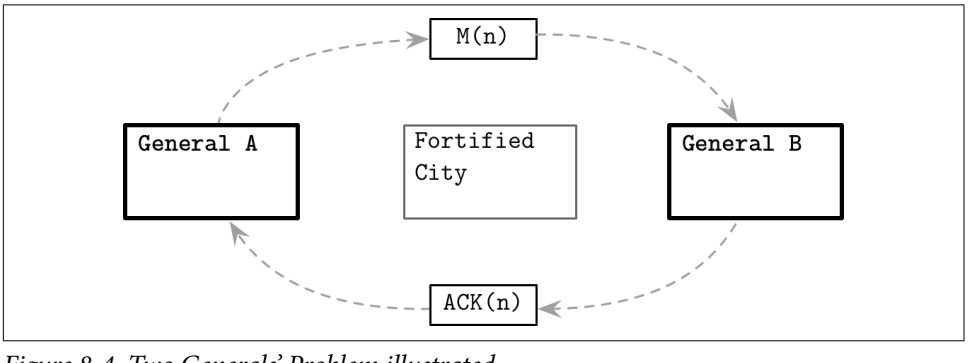

Figure 8-4. Two Generals' Problem illustrated

The messenger carrying this acknowledgment might get captured or fail to deliver it, as well. `B` doesn't have any way of knowing if the messenger has successfully delivered the acknowledgment.

To be sure about it, `B` has to wait for `ACK(ACK(MSG(N)))`, a second-order acknowledgment stating that `A` received an acknowledgment for the acknowledgment.

No matter how many further confirmations the generals send to each other, they will always be one `ACK` away from knowing if they can safely proceed with the attack. The generals are doomed to wonder if the message carrying this last acknowledgment has reached the destination.

Notice that we did not make any timing assumptions: communication between generals is fully asynchronous. There is no upper time bound set on how long the generals can take to respond.

FLP Impossibility

In a paper by Fisher, Lynch, and Paterson, the authors describe a problem famously known as the *FLP Impossibility Problem* [FISCHER85] (derived from the first letters of authors' last names), wherein they discuss a form of consensus in which processes start with an initial value and attempt to agree on a new value. After the algorithm completes, this new value has to be the same for all nonfaulty processes.

Reaching an agreement on a specific value is straightforward if the network is entirely reliable; but in reality, systems are prone to many different sorts of failures, such as message loss, duplication, network partitions, and slow or crashed processes.

A consensus protocol describes a system that, given multiple processes starting at its *initial state*, brings all of the processes to the *decision state*. For a consensus protocol to be correct, it has to preserve three properties:

Agreement
: The decision the protocol arrives at has to be unanimous: each process decides on some value, and this has to be the same for all processes. Otherwise, we have not reached a consensus.

Validity
: The agreed value has to be *proposed* by one of the participants, which means that the system should not just "come up" with the value. This also implies nontriviality of the value: processes should not always decide on some predefined default value.

Termination
: An agreement is final only if there are no processes that did not reach the decision state.

[FISCHER85] assumes that processing is entirely asynchronous; there's no shared notion of time between the processes. Algorithms in such systems cannot be based on timeouts, and there's no way for a process to find out whether the other process has crashed or is simply running too slow. The paper shows that, given these assumptions, there exists no protocol that can guarantee consensus in a bounded time. No completely asynchronous consensus algorithm can tolerate the unannounced crash of even a single remote process.

If we do not consider an upper time bound for the process to complete the algorithm steps, process failures can't be reliably detected, and there's no deterministic algorithm to reach a consensus.

However, FLP Impossibility does not mean we have to pack our things and go home, as reaching consensus is not possible. It only means that we cannot always reach consensus in an asynchronous system in bounded time. In practice, systems exhibit at least some degree of synchrony, and the solution to this problem requires a more refined model.

System Synchrony

From FLP Impossibility, you can see that the timing assumption is one of the critical characteristics of the distributed system. In an *asynchronous system*, we do not know the relative speeds of processes, and cannot guarantee message delivery in a bounded time or a particular order. The process might take indefinitely long to respond, and process failures can't always be reliably detected.

The main criticism of asynchronous systems is that these assumptions are not realistic: processes can't have *arbitrarily* different processing speeds, and links don't take *indefinitely* long to deliver messages. Relying on time both simplifies reasoning and helps to provide upper-bound timing guarantees.

It is not always possible to solve a consensus problem in an asynchronous model [FISCHER85]. Moreover, designing an efficient synchronous algorithm is not always achievable, and for some tasks the practical solutions are more likely to be time-dependent [ARJOMANDI83].

These assumptions can be loosened up, and the system can be considered to be *synchronous*. For that, we introduce the notion of timing. It is much easier to reason about the system under the synchronous model. It assumes that processes are progressing at comparable rates, that transmission delays are bounded, and message delivery cannot take arbitrarily long.

A synchronous system can also be represented in terms of synchronized process-local clocks: there is an upper time bound in time difference between the two process-local time sources [CACHIN11].

Designing systems under a synchronous model allows us to use timeouts. We can build more complex abstractions, such as leader election, consensus, failure detection, and many others on top of them. This makes the best-case scenarios more robust, but results in a failure if the timing assumptions don't hold up. For example, in the Raft consensus algorithm (see "Raft" on page 300), we may end up with multiple processes believing they're leaders, which is resolved by forcing the lagging process to accept the other process as a leader; failure-detection algorithms (see Chapter 9) can wrongly identify a live process as failed or vice versa. When designing our systems, we should make sure to consider these possibilities.

Properties of both asynchronous and synchronous models can be combined, and we can think of a system as *partially synchronous*. A partially synchronous system exhibits some of the properties of the synchronous system, but the bounds of message delivery, clock drift, and relative processing speeds might not be exact and hold only *most of the time* [DWORK88].

Synchrony is an essential property of the distributed system: it has an impact on performance, scalability, and general solvability, and has many factors necessary for the correct functioning of our systems. Some of the algorithms we discuss in this book operate under the assumptions of synchronous systems.

Failure Models

We keep mentioning *failures*, but so far it has been a rather broad and generic concept that might capture many meanings. Similar to how we can make different timing assumptions, we can assume the presence of different types of failures. A *failure model* describes exactly how processes can crash in a distributed system, and algorithms are developed using these assumptions. For example, we can assume that a process can crash and never recover, or that it is expected to recover after some time passes, or that it can fail by spinning out of control and supplying incorrect values.

In distributed systems, processes rely on one another for executing an algorithm, so failures can result in incorrect execution across the whole system.

We'll discuss multiple failure models present in distributed systems, such as *crash*, *omission*, and *arbitrary* faults. This list is not exhaustive, but it covers most of the cases applicable and important in real-life systems.

Crash Faults

Normally, we expect the process to be executing all steps of an algorithm correctly. The simplest way for a process to crash is by *stopping* the execution of any further steps required by the algorithm and not sending any messages to other processes. In other words, the process *crashes*. Most of the time, we assume a *crash-stop* process abstraction, which prescribes that, once the process has crashed, it remains in this state.

This model does not assume that it is impossible for the process to recover, and does not discourage recovery or try to prevent it. It only means that the algorithm *does not rely* on recovery for correctness or liveness. Nothing prevents processes from recovering, catching up with the system state, and participating in the *next* instance of the algorithm.

Failed processes are not able to continue participating in the current round of negotiations during which they failed. Assigning the recovering process a new, different

identity does not make the model equivalent to crash-recovery (discussed next), since most algorithms use predefined lists of processes and clearly define failure semantics in terms of how many failures they can tolerate [CACHIN11].

Crash-recovery is a different process abstraction, under which the process stops executing the steps required by the algorithm, but recovers at a later point and tries to execute further steps. The possibility of recovery requires introducing a durable state and recovery protocol into the system [SKEEN83]. Algorithms that allow crash-recovery need to take all possible recovery states into consideration, since the recovering process may attempt to continue execution from the last step known to it.

Algorithms, aiming to exploit recovery, have to take both state and identity into account. Crash-recovery, in this case, can also be viewed as a special case of omission failure, since from the other process's perspective there's no distinction between the process that was unreachable and the one that has crashed and recovered.

Omission Faults

Another failure model is *omission fault*. This model assumes that the process skips some of the algorithm steps, or is not able to execute them, or this execution is not visible to other participants, or it cannot send or receive messages to and from other participants. Omission fault captures network partitions between the processes caused by faulty network links, switch failures, or network congestion. Network partitions can be represented as omissions of messages between individual processes or process groups. A crash can be simulated by completely omitting any messages to and from the process.

When the process is operating slower than the other participants and sends responses much later than expected, for the rest of the system it may look like it is forgetful. Instead of stopping completely, a slow node attempts to send its results out of sync with other nodes.

Omission failures occur when the algorithm that was supposed to execute certain steps either skips them or the results of this execution are not visible. For example, this may happen if the message is lost on the way to the recipient, and the sender fails to send it again and continues to operate as if it was successfully delivered, even though it was irrecoverably lost. Omission failures can also be caused by intermittent hangs, overloaded networks, full queues, etc.

Arbitrary Faults

The hardest class of failures to overcome is *arbitrary* or *Byzantine* faults: a process continues executing the algorithm steps, but in a way that contradicts the algorithm (for example, if a process in a consensus algorithm decides on a value that no other participant has ever proposed).

Such failures can happen due to bugs in software, or due to processes running different versions of the algorithm, in which case failures are easier to find and understand. It can get much more difficult when we do not have control over all processes, and one of the processes is intentionally misleading other processes.

You might have heard of Byzantine fault tolerance from the airspace industry: airplane and spacecraft systems do not take responses from subcomponents at face value and cross-validate their results. Another widespread application is cryptocurrencies [GILAD17], where there is no central authority, different parties control the nodes, and adversary participants have a material incentive to forge values and attempt to game the system by providing faulty responses.

Handling Failures

We can *mask* failures by forming process groups and introducing redundancy into the algorithm: even if one of the processes fails, the user will not notice this failure [CHRISTIAN91].

There might be some performance penalty related to failures: normal execution relies on processes being responsive, and the system has to fall back to the slower execution path for error handling and correction. Many failures can be prevented on the software level by code reviews, extensive testing, ensuring message delivery by introducing timeouts and retries, and making sure that steps are executed in order locally.

Most of the algorithms we're going to cover here assume the crash-failure model and work around failures by introducing redundancy. These assumptions help to create algorithms that perform better and are easier to understand and implement.

Summary

In this chapter, we discussed some of the distributed systems terminology and introduced some basic concepts. We've talked about the inherent difficulties and complications caused by the unreliability of the system components: links may fail to deliver messages, processes may crash, or the network may get partitioned.

This terminology should be enough for us to continue the discussion. The rest of the book talks about the *solutions* commonly used in distributed systems: we think back to what can go wrong and see what options we have available.

Further Reading

If you'd like to learn more about the concepts mentioned in this chapter, you can refer to the following sources:

Distributed systems abstractions, failure models, and timing assumptions

Lynch, Nancy A. 1996. *Distributed Algorithms*. San Francisco: Morgan Kaufmann.

Tanenbaum, Andrew S. and Maarten van Steen. 2006. *Distributed Systems: Principles and Paradigms* (2nd Ed). Boston: Pearson.

Cachin, Christian, Rachid Guerraoui, and Lus Rodrigues. 2011. *Introduction to Reliable and Secure Distributed Programming* (2nd Ed.). New York: Springer.

CHAPTER 9
Failure Detection

> If a tree falls in a forest and no one is around to hear it, does it make a sound?
>
> —Unknown Author

In order for a system to appropriately react to failures, failures should be detected in a timely manner. A faulty process might get contacted even though it won't be able to respond, increasing latencies and reducing overall system availability.

Detecting failures in asynchronous distributed systems (i.e., without making any timing assumptions) is extremely difficult as it's impossible to tell whether the process has crashed, or is running slowly and taking an indefinitely long time to respond. We discussed a problem related to this one in "FLP Impossibility" on page 189.

Terms such as *dead*, *failed*, and *crashed* are usually used to describe a process that has stopped executing its steps completely. Terms such as *unresponsive*, *faulty*, and *slow* are used to describe *suspected* processes, which may actually be dead.

Failures may occur on the *link* level (messages between processes are lost or delivered slowly), or on the *process* level (the process crashes or is running slowly), and slowness may not always be distinguishable from failure. This means there's always a trade-off between wrongly suspecting alive processes as dead (producing *false-positives*), and delaying marking an unresponsive process as dead, giving it the benefit of doubt and expecting it to respond eventually (producing *false-negatives*).

A *failure detector* is a local subsystem responsible for identifying failed or unreachable processes to exclude them from the algorithm and guarantee liveness while preserving safety.

Liveness and safety are the properties that describe an algorithm's ability to solve a specific problem and the correctness of its output. More formally, *liveness* is a property that guarantees that a specific intended event *must* occur. For example, if one of

the processes has failed, a failure detector *must* detect that failure. *Safety* guarantees that unintended events will *not* occur. For example, if a failure detector has marked a process as dead, this process had to be, in fact, dead [LAMPORT77] [RAYNAL99] [FREILING11].

From a practical perspective, excluding failed processes helps to avoid unnecessary work and prevents error propagation and cascading failures, while reducing availability when excluding potentially suspected alive processes.

Failure-detection algorithms should exhibit several essential properties. First of all, every nonfaulty member should eventually notice the process failure, and the algorithm should be able to make progress and eventually reach its final result. This property is called *completeness*.

We can judge the quality of the algorithm by its *efficiency*: how fast the failure detector can identify process failures. Another way to do this is to look at the *accuracy* of the algorithm: whether or not the process failure was precisely detected. In other words, an algorithm is *not* accurate if it falsely accuses a live process of being failed or is not able to detect the existing failures.

We can think of the relationship between efficiency and accuracy as a tunable parameter: a more efficient algorithm might be less precise, and a more accurate algorithm is usually less efficient. It is provably impossible to build a failure detector that is both accurate and efficient. At the same time, failure detectors are allowed to produce false-positives (i.e., falsely identify live processes as failed and vice versa) [CHANDRA96].

Failure detectors are an essential prerequisite and an integral part of many consensus and atomic broadcast algorithms, which we'll be discussing later in this book.

Many distributed systems implement failure detectors by using *heartbeats*. This approach is quite popular because of its simplicity and strong completeness. Algorithms we discuss here assume the absence of Byzantine failures: processes do not attempt to intentionally lie about their state or states of their neighbors.

Heartbeats and Pings

We can query the state of remote processes by triggering one of two periodic processes:

- We can trigger a ping, which sends messages to remote processes, checking if they are still alive by expecting a response within a specified time period.
- We can trigger a *heartbeat* when the process is actively notifying its peers that it's still running by sending messages to them.

We'll use pings as an example here, but the same problem can be solved using heartbeats, producing similar results.

Each process maintains a list of other processes (alive, dead, and suspected ones) and updates it with the last response time for each process. If a process fails to respond to a ping message for a longer time, it is marked as *suspected*.

Figure 9-1 shows the normal functioning of a system: process `P1` is querying the state of neighboring node `P2`, which responds with an acknowledgment.

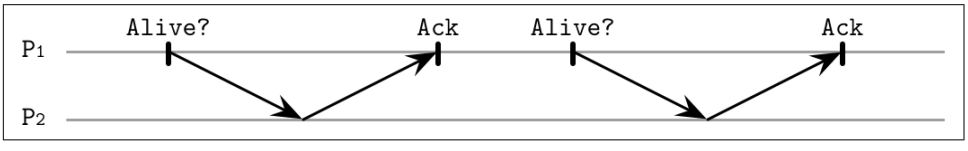

Figure 9-1. Pings for failure detection: normal functioning, no message delays

In contrast, Figure 9-2 shows how acknowledgment messages are delayed, which might result in marking the active process as down.

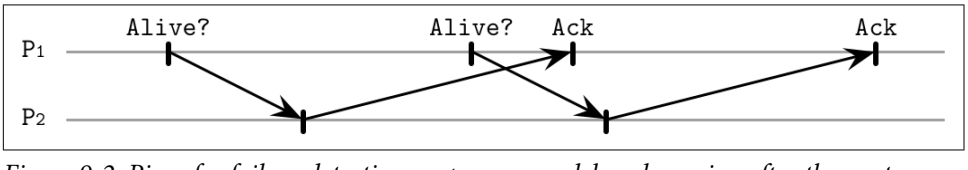

Figure 9-2. Pings for failure detection: responses are delayed, coming after the next message is sent

Many failure-detection algorithms are based on heartbeats and timeouts. For example, Akka, a popular framework for building distributed systems, has an implementation of a deadline failure detector, which uses heartbeats and reports a process failure if it has failed to register within a fixed time interval.

This approach has several potential downsides: its precision relies on the careful selection of ping frequency and timeout, and it does not capture process visibility from the perspective of other processes (see "Outsourced Heartbeats" on page 198).

Timeout-Free Failure Detector

Some algorithms avoid relying on timeouts for detecting failures. For example, Heartbeat, a *timeout-free* failure detector [AGUILERA97], is an algorithm that only counts heartbeats and allows the application to detect process failures based on the data in the heartbeat counter vectors. Since this algorithm is timeout-free, it operates under *asynchronous* system assumptions.

The algorithm assumes that any two correct processes are connected to each other with a *fair path*, which contains only fair links (i.e., if a message is sent over this link

infinitely often, it is also received infinitely often), and each process is aware of the existence of *all* other processes in the network.

Each process maintains a list of neighbors and counters associated with them. Processes start by sending heartbeat messages to their neighbors. Each message contains a path that the heartbeat has traveled so far. The initial message contains the first sender in the path and a unique identifier that can be used to avoid broadcasting the same message multiple times.

When the process receives a new heartbeat message, it increments counters for all participants present in the path and sends the heartbeat to the ones that are not present there, appending itself to the path. Processes stop propagating messages as soon as they see that all the known processes have already received it (in other words, process IDs appear in the path).

Since messages are propagated through different processes, and heartbeat paths contain aggregated information received from the neighbors, we can (correctly) mark an unreachable process as alive even when the direct link between the two processes is faulty.

Heartbeat counters represent a global and normalized view of the system. This view captures how the heartbeats are propagated relative to one another, allowing us to compare processes. However, one of the shortcomings of this approach is that interpreting heartbeat counters may be quite tricky: we need to pick a threshold that can yield reliable results. Unless we can do that, the algorithm will falsely mark active processes as suspected.

Outsourced Heartbeats

An alternative approach, used by the Scalable Weakly Consistent Infection-style Process Group Membership Protocol (SWIM) [GUPTA01] is to use *outsourced heartbeats* to improve reliability using information about the process liveness from the perspective of its neighbors. This approach does not require processes to be aware of all other processes in the network, only a subset of connected peers.

As shown in Figure 9-3, process P_1 sends a ping message to process P_2. P_2 doesn't respond to the message, so P_1 proceeds by selecting multiple random members (P_3 and P_4). These random members try sending heartbeat messages to P_2 and, if it responds, forward acknowledgments back to P_1.

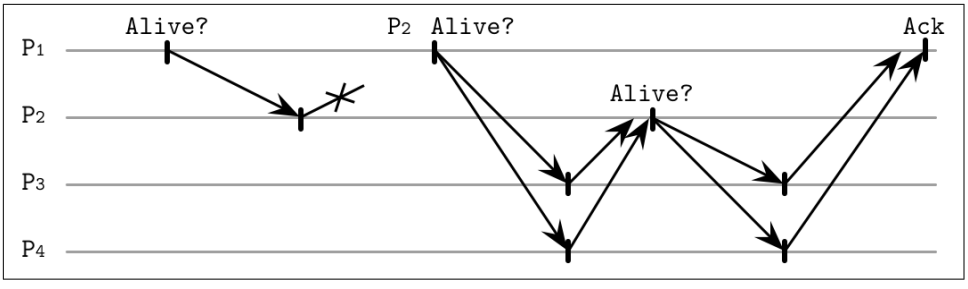

Figure 9-3. "Outsourcing" heartbeats

This allows accounting for both direct and indirect reachability. For example, if we have processes P_1, P_2, and P_3, we can check the state of P_3 from the perspective of both P_1 and P_2.

Outsourced heartbeats allow reliable failure detection by distributing responsibility for deciding across the group of members. This approach does not require broadcasting messages to a broad group of peers. Since outsourced heartbeat requests can be triggered in parallel, this approach can collect more information about suspected processes quickly, and allow us to make more accurate decisions.

Phi-Accural Failure Detector

Instead of treating node failure as a binary problem, where the process can be only in two states: up or down, a *phi-accrual* (φ-accrual) failure detector [HAYASHIBARA04] has a continuous scale, capturing the probability of the monitored process's crash. It works by maintaining a sliding window, collecting arrival times of the most recent heartbeats from the peer processes. This information is used to approximate arrival time of the *next* heartbeat, compare this approximation with the actual arrival time, and compute the *suspicion level* φ: how certain the failure detector is about the failure, given the current network conditions.

The algorithm works by collecting and sampling arrival times, creating a view that can be used to make a reliable judgment about node health. It uses these samples to compute the value of φ: if this value reaches a threshold, the node is marked as down. This failure detector dynamically adapts to changing network conditions by adjusting the scale on which the node can be marked as a suspect.

From the architecture perspective, a phi-accrual failure detector can be viewed as a combination of three subsystems:

Monitoring
: Collecting liveness information through pings, heartbeats, or request-response sampling.

Interpretation
: Making a decision on whether or not the process should be marked as suspected.

Action
: A callback executed whenever the process is marked as suspected.

The monitoring process collects and stores data samples (which are assumed to follow a normal distribution) in a fixed-size window of heartbeat arrival times. Newer arrivals are added to the window, and the oldest heartbeat data points are discarded.

Distribution parameters are estimated from the sampling window by determining the mean and variance of samples. This information is used to compute the probability of arrival of the message within `t` time units after the previous one. Given this information, we compute φ, which describes how likely we are to make a correct decision about a process's liveness. In other words, how likely it is to make a mistake and receive a heartbeat that will contradict the calculated assumptions.

This approach was developed by researchers from the Japan Advanced Institute of Science and Technology, and is now used in many distributed systems; for example, Cassandra and Akka (along with the aforementioned deadline failure detector).

Gossip and Failure Detection

Another approach that avoids relying on a single-node view to make a decision is a gossip-style failure detection service [VANRENESSE98], which uses *gossip* (see "Gossip Dissemination" on page 250) to collect and distribute states of neighboring processes.

Each member maintains a list of other members, their *heartbeat counters*, and timestamps, specifying when the heartbeat counter was incremented for the last time. Periodically, each member increments its heartbeat counter and distributes its list to a random neighbor. Upon the message receipt, the neighboring node merges the list with its own, updating heartbeat counters for the other neighbors.

Nodes also periodically check the list of states and heartbeat counters. If any node did not update its counter for long enough, it is considered failed. This timeout period should be chosen carefully to minimize the probability of false-positives. How often members have to communicate with each other (in other words, worst-case bandwidth) is capped, and can grow at most linearly with a number of processes in the system.

Figure 9-4 shows three communicating processes sharing their heartbeat counters:

- a) All three can communicate and update their timestamps.
- b) `P3` isn't able to communicate with `P1`, but its timestamp t_6 can still be propagated through `P2`.
- c) `P3` crashes. Since it doesn't send updates anymore, it is detected as failed by other processes.

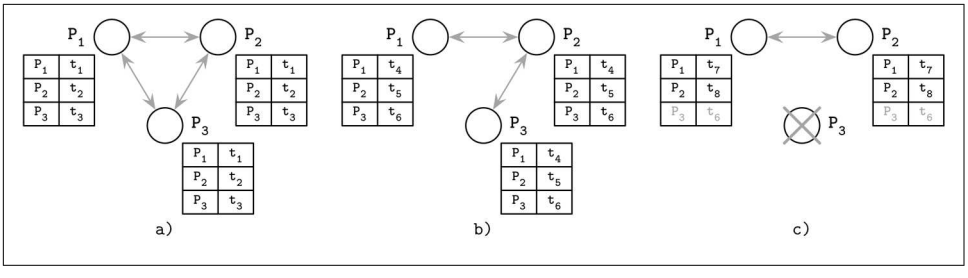

Figure 9-4. Replicated heartbeat table for failure detection

This way, we can detect crashed nodes, as well as the nodes that are unreachable by any other cluster member. This decision is reliable, since the view of the cluster is an aggregate from multiple nodes. If there's a link failure between the two hosts, heartbeats can still propagate through other processes. Using gossip for propagating system states increases the number of messages in the system, but allows information to spread more reliably.

Reversing Failure Detection Problem Statement

Since propagating the information about failures is not always possible, and propagating it by notifying every member might be expensive, one of the approaches, called *FUSE* (failure notification service) [DUNAGAN04], focuses on reliable and cheap failure propagation that works even in cases of network partitions.

To detect process failures, this approach arranges all active processes in groups. If one of the groups becomes unavailable, all participants detect the failure. In other words, every time a single process failure is detected, it is converted and propagated as a *group failure*. This allows detecting failures in the presence of any pattern of disconnects, partitions, and node failures.

Processes in the group periodically send ping messages to other members, querying whether they're still alive. If one of the members cannot respond to this message because of a crash, network partition, or link failure, the member that has initiated this ping will, in turn, stop responding to ping messages itself.

Figure 9-5 shows four communicating processes:

- a) Initial state: all processes are alive and can communicate.
- b) P_2 crashes and stops responding to ping messages.
- c) P_4 detects the failure of P_2 and stops responding to ping messages itself.
- d) Eventually, P_1 and P_3 notice that both P_1 and P_2 do not respond, and process failure propagates to the entire group.

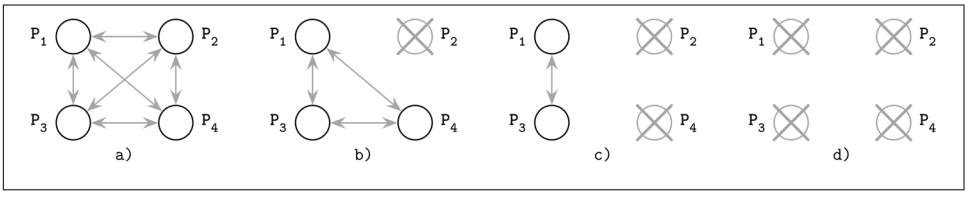

Figure 9-5. FUSE failure detection

All failures are propagated through the system from the source of failure to all other participants. Participants gradually stop responding to pings, converting from the individual node failure to the group failure.

Here, we use the absence of communication as a means of propagation. An advantage of using this approach is that every member is guaranteed to learn about group failure and adequately react to it. One of the downsides is that a link failure separating a single process from other ones can be converted to the group failure as well, but this can be seen as an advantage, depending on the use case. Applications can use their own definitions of propagated failures to account for this scenario.

Summary

Failure detectors are an essential part of any distributed system. As shown by the FLP Impossibility result, no protocol can guarantee consensus in an asynchronous system. Failure detectors help to augment the model, allowing us to solve a consensus problem by making a trade-off between accuracy and completeness. One of the significant findings in this area, proving the usefulness of failure detectors, was described in [CHANDRA96], which shows that solving consensus is possible even with a failure detector that makes an infinite number of mistakes.

We've covered several algorithms for failure detection, each using a different approach: some focus on detecting failures by direct communication, some use broadcast or gossip for spreading the information around, and some opt out by using quiescence (in other words, absence of communication) as a means of propagation. We now know that we can use heartbeats or pings, hard deadlines, or continuous

scales. Each one of these approaches has its own upsides: simplicity, accuracy, or precision.

Further Reading

If you'd like to learn more about the concepts mentioned in this chapter, you can refer to the following sources:

Failure detection and algorithms

Chandra, Tushar Deepak and Sam Toueg. 1996. "Unreliable failure detectors for reliable distributed systems." *Journal of the ACM* 43, no. 2 (March): 225-267. *https://doi.org/10.1145/226643.226647*.

Freiling, Felix C., Rachid Guerraoui, and Petr Kuznetsov. 2011. "The failure detector abstraction." *ACM Computing Surveys* 43, no. 2 (January): Article 9. *https://doi.org/10.1145/1883612.1883616*.

Phan-Ba, Michael. 2015. "A literature review of failure detection within the context of solving the problem of distributed consensus." *https://www.cs.ubc.ca/~bestchai/theses/michael-phan-ba-msc-essay-2015.pdf*

CHAPTER 10
Leader Election

Synchronization can be quite costly: if each algorithm step involves contacting each other participant, we can end up with a significant communication overhead. This is particularly true in large and geographically distributed networks. To reduce synchronization overhead and the number of message round-trips required to reach a decision, some algorithms rely on the existence of the *leader* (sometimes called *coordinator*) process, responsible for executing or coordinating steps of a distributed algorithm.

Generally, processes in distributed systems are uniform, and any process can take over the leadership role. Processes assume leadership for long periods of time, but this is not a permanent role. Usually, the process remains a leader until it crashes. After the crash, any other process can start a new election round, assume leadership, if it gets elected, and continue the failed leader's work.

The *liveness* of the election algorithm guarantees that *most of the time* there will be a leader, and the election will eventually complete (i.e., the system should not be in the election state indefinitely).

Ideally, we'd like to assume *safety*, too, and guarantee there may be *at most one* leader at a time, and completely eliminate the possibility of a *split brain* situation (when two leaders serving the same purpose are elected but unaware of each other). However, in practice, many leader election algorithms violate this agreement.

Leader processes can be used, for example, to achieve a total order of messages in a broadcast. The leader collects and holds the global state, receives messages, and disseminates them among the processes. It can also be used to coordinate system reorganization after the failure, during initialization, or when important state changes happen.

Election is triggered when the system initializes, and the leader is elected for the first time, or when the previous leader crashes or fails to communicate. Election has to be deterministic: exactly one leader has to emerge from the process. This decision needs to be effective for all participants.

Even though leader election and distributed locking (i.e., exclusive ownership over a shared resource) might look alike from a theoretical perspective, they are slightly different. If one process holds a lock for executing a critical section, it is unimportant for other processes to know who exactly is holding a lock right now, as long as the liveness property is satisfied (i.e., the lock will be eventually released, allowing others to acquire it). In contrast, the elected process has some special properties and has to be known to all other participants, so the newly elected leader has to notify its peers about its role.

If a distributed locking algorithm has any sort of preference toward some process or group of processes, it will eventually starve nonpreferred processes from the shared resource, which contradicts the liveness property. In contrast, the leader can remain in its role until it stops or crashes, and long-lived leaders are preferred.

Having a stable leader in the system helps to avoid state synchronization between remote participants, reduce the number of exchanged messages, and drive execution from a single process instead of requiring peer-to-peer coordination. One of the potential problems in systems with a notion of leadership is that the leader can become a bottleneck. To overcome that, many systems partition data in nonintersecting independent replica sets (see "Database Partitioning" on page 270). Instead of having a single system-wide leader, each replica set has its own leader. One of the systems that uses this approach is Spanner (see "Distributed Transactions with Spanner" on page 268).

Because every leader process will eventually fail, failure has to be detected, reported, and reacted upon: a system has to elect another leader to replace the failed one.

Some algorithms, such as ZAB (see "Zookeeper Atomic Broadcast (ZAB)" on page 283), Multi-Paxos (see "Multi-Paxos" on page 291), or Raft (see "Raft" on page 300), use temporary leaders to reduce the number of messages required to reach an agreement between the participants. However, these algorithms use their own algorithm-specific means for leader election, failure detection, and resolving conflicts between the competing leader processes.

Bully Algorithm

One of the leader election algorithms, known as the *bully algorithm*, uses process ranks to identify the new leader. Each process gets a unique rank assigned to it. During the election, the process with the highest rank becomes a leader [MOLINA82].

This algorithm is known for its simplicity. The algorithm is named *bully* because the highest-ranked node "bullies" other nodes into accepting it. It is also known as *monarchial* leader election: the highest-ranked sibling becomes a monarch after the previous one ceases to exist.

Election starts if one of the processes notices that there's no leader in the system (it was never initialized) or the previous leader has stopped responding to requests, and proceeds in three steps:[1]

1. The process sends election messages to processes with higher identifiers.
2. The process waits, allowing higher-ranked processes to respond. If no higher-ranked process responds, it proceeds with step 3. Otherwise, the process notifies the highest-ranked process it has heard from, and allows it to proceed with step 3.
3. The process assumes that there are no active processes with a higher rank, and notifies all lower-ranked processes about the new leader.

Figure 10-1 illustrates the bully leader election algorithm:

- a) Process `3` notices that the previous leader `6` has crashed and starts a new election by sending `Election` messages to processes with higher identifiers.
- b) `4` and `5` respond with `Alive`, as they have a higher rank than `3`.
- c) `3` notifies the highest-ranked process `5` that has responded during this round.
- d) `5` is elected as a new leader. It broadcasts `Elected` messages, notifying lower-ranked processes about the election results.

1 These steps describe the *modified* bully election algorithm [KORDAFSHARI05] as it's more compact and clear.

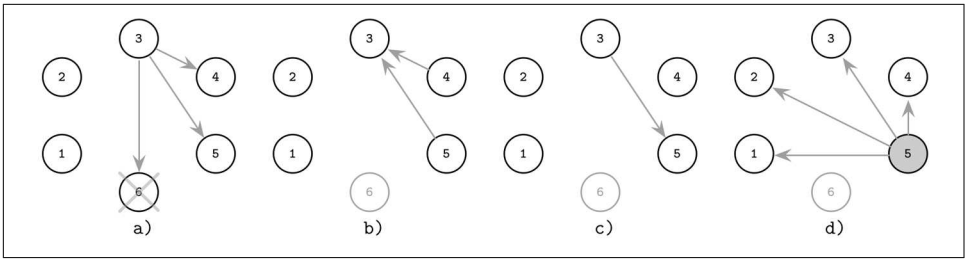

Figure 10-1. Bully algorithm: previous leader (6) fails and process 3 starts the new election

One of the apparent problems with this algorithm is that it violates the safety guarantee (that at most one leader can be elected at a time) in the presence of network partitions. It is quite easy to end up in the situation where nodes get split into two or more independently functioning subsets, and each subset elects its leader. This situation is called *split brain*.

Another problem with this algorithm is a strong preference toward high-ranked nodes, which becomes an issue if they are unstable and can lead to a permanent state of reelection. An unstable high-ranked node proposes itself as a leader, fails shortly thereafter, wins reelection, fails again, and the whole process repeats. This problem can be solved by distributing host quality metrics and taking them into consideration during the election.

Next-In-Line Failover

There are many versions of the bully algorithm that improve its various properties. For example, we can use multiple next-in-line alternative processes as a failover to shorten reelections [GHOLIPOUR09].

Each elected leader provides a list of failover nodes. When one of the processes detects a leader failure, it starts a new election round by sending a message to the highest-ranked *alternative* from the list provided by the failed leader. If one of the proposed alternatives is up, it becomes a new leader without having to go through the complete election round.

If the process that has detected the leader failure is itself the highest ranked process from the list, it can notify the processes about the new leader right away.

Figure 10-2 shows the process with this optimization in place:

- a) `6`, a leader with designated alternatives `{5,4}`, crashes. `3` notices this failure and contacts `5`, the alternative from the list with the highest rank.

- b) 5 responds to 3 that it's alive to prevent it from contacting other nodes from the alternatives list.
- c) 5 notifies other nodes that it's a new leader.

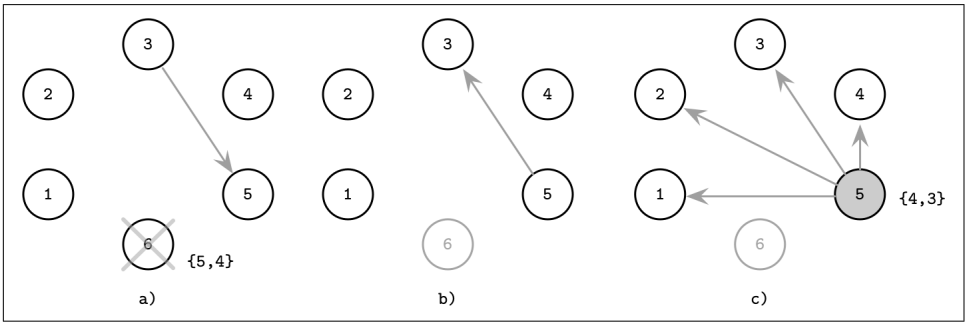

Figure 10-2. Bully algorithm with failover: previous leader (6) fails and process 3 starts the new election by contacting the highest-ranked alternative

As a result, we require fewer steps during the election if the next-in-line process is alive.

Candidate/Ordinary Optimization

Another algorithm attempts to lower requirements on the number of messages by splitting the nodes into two subsets, *candidate* and *ordinary*, where only one of the candidate nodes can eventually become a leader [MURSHED12].

The ordinary process initiates election by contacting candidate nodes, collecting responses from them, picking the highest-ranked alive candidate as a new leader, and then notifying the rest of the nodes about the election results.

To solve the problem with multiple simultaneous elections, the algorithm proposes to use a tiebreaker variable δ, a process-specific delay, varying significantly between the nodes, that allows one of the nodes to initiate the election before the other ones. The tiebreaker time is generally greater than the message round-trip time. Nodes with higher priorities have a lower δ, and vice versa.

Figure 10-3 shows the steps of the election process:

- a) Process 4 from the ordinary set notices the failure of leader process 6. It starts a new election round by contacting all remaining processes from the candidate set.
- b) Candidate processes respond to notify 4 that they're still alive.
- c) 4 notifies all processes about the new leader: 2.

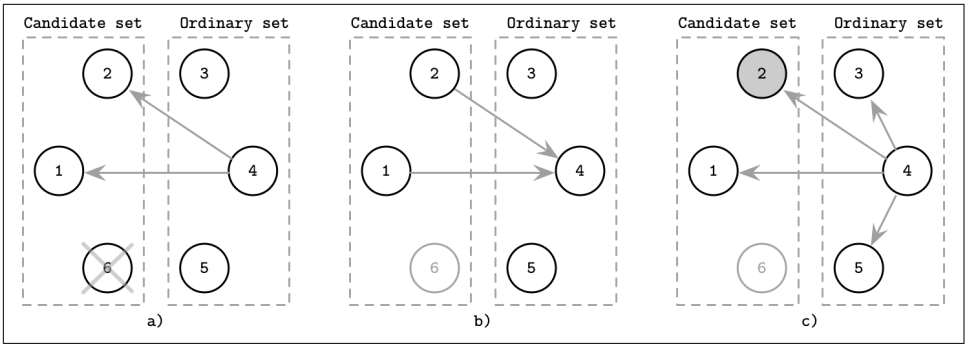

Figure 10-3. Candidate/ordinary modification of the bully algorithm: previous leader (6) fails and process 4 starts the new election

Invitation Algorithm

An *invitation algorithm* allows processes to "invite" other processes to join their groups instead of trying to outrank them. This algorithm allows multiple leaders *by definition*, since each group has its own leader.

Each process starts as a leader of a new group, where the only member is the process itself. Group leaders contact peers that do not belong to their groups, inviting them to join. If the peer process is a leader itself, two groups are merged. Otherwise, the contacted process responds with a group leader ID, allowing two group leaders to establish contact and merge groups in fewer steps.

Figure 10-4 shows the execution steps of the invitation algorithm:

- a) Four processes start as leaders of groups containing one member each. `1` invites `2` to join its group, and `3` invites `4` to join its group.
- b) `2` joins a group with process `1`, and `4` joins a group with process `3`. `1`, the leader of the first group, contacts `3`, the leader of the other group. Remaining group members (`4`, in this case) are notified about the new group leader.
- c) Two groups are merged and `1` becomes a leader of an extended group.

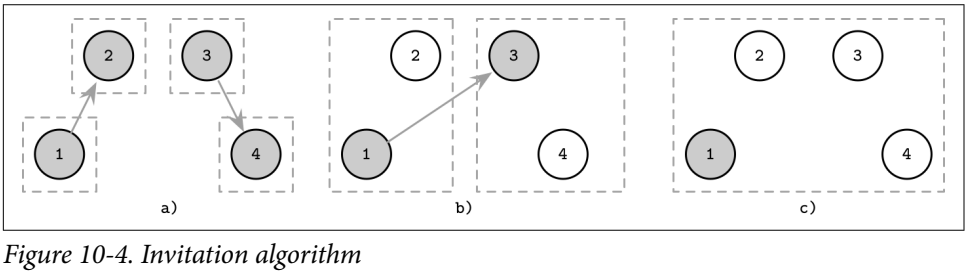

Figure 10-4. Invitation algorithm

Since groups are merged, it doesn't matter whether the process that suggested the group merge becomes a new leader or the other one does. To keep the number of messages required to merge groups to a minimum, a leader of a larger group can become a leader for a new group. This way only the processes from the smaller group have to be notified about the change of leader.

Similar to the other discussed algorithms, this algorithm allows processes to settle in multiple groups and have multiple leaders. The invitation algorithm allows creating process groups and merging them without having to trigger a new election from scratch, reducing the number of messages required to finish the election.

Ring Algorithm

In the ring algorithm [CHANG79], all nodes in the system form a ring and are aware of the ring topology (i.e., their predecessors and successors in the ring). When the process detects the leader failure, it starts the new election. The election message is forwarded across the ring: each process contacts its successor (the next node closest to it in the ring). If this node is unavailable, the process skips the unreachable node and attempts to contact the nodes after it in the ring, until eventually one of them responds.

Nodes contact their siblings, following around the ring and collecting the live node set, adding themselves to the set before passing it over to the next node, similar to the failure-detection algorithm described in "Timeout-Free Failure Detector" on page 197, where nodes append their identifiers to the path before passing it to the next node.

The algorithm proceeds by fully traversing the ring. When the message comes back to the node that started the election, the highest-ranked node from the live set is chosen as a leader. In Figure 10-5, you can see an example of such a traversal:

- a) Previous leader `6` has failed and each process has a view of the ring from its perspective.
- b) `3` initiates an election round by starting traversal. On each step, there's a set of nodes traversed on the path so far. `5` can't reach `6`, so it skips it and goes straight to `1`.
- c) Since `5` was the node with the highest rank, `3` initiates another round of messages, distributing the information about the new leader.

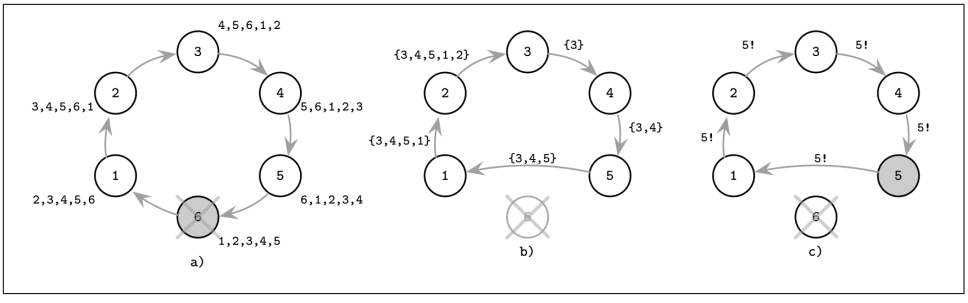

Figure 10-5. Ring algorithm: previous leader (6) fails and 3 starts the election process

Variants of this algorithm include collecting a single highest-ranked identifier instead of a set of active nodes to save space: since the `max` function is commutative, it is enough to know a current maximum. When the algorithm comes back to the node that has started the election, the last known highest identifier is circulated across the ring once again.

Since the ring can be partitioned in two or more parts, with each part potentially electing its own leader, this approach doesn't hold a safety property, either.

As you can see, for a system with a leader to function correctly, we need to know the status of the current leader (whether it is alive or not), since to keep processes organized and for execution to continue, the leader has to be alive and reachable to perform its duties. To detect leader crashes, we can use failure-detection algorithms (see Chapter 9).

Summary

Leader election is an important subject in distributed systems, since using a designated leader helps to reduce coordination overhead and improve the algorithm's performance. Election rounds might be costly but, since they're infrequent, they do not have a negative impact on the overall system performance. A single leader can become a bottleneck, but most of the time this is solved by partitioning data and using per-partition leaders or using different leaders for different actions.

Unfortunately, all the algorithms we've discussed in this chapter are prone to the split brain problem: we can end up with two leaders in independent subnets that are not aware of each other's existence. To avoid split brain, we have to obtain a cluster-wide majority of votes.

Many consensus algorithms, including Multi-Paxos and Raft, rely on a leader for coordination. But isn't leader election the same as consensus? To elect a leader, we need to reach a consensus about its identity. If we can reach consensus about the

leader identity, we can use the same means to reach consensus on anything else [ABRAHAM13].

The identity of a leader may change without processes knowing about it, so the question is whether the process-local knowledge about the leader is still valid. To achieve that, we need to combine leader election with failure detection. For example, the *stable leader election* algorithm uses rounds with a unique stable leader and timeout-based failure detection to guarantee that the leader can retain its position for as long as it doesn't crash and is accessible [AGUILERA01].

Algorithms that rely on leader election often *allow* the existence of multiple leaders and attempt to resolve conflicts between the leaders as quickly as possible. For example, this is true for Multi-Paxos (see "Multi-Paxos" on page 291), where only one of the two conflicting leaders (proposers) can proceed, and these conflicts are resolved by collecting a second quorum, guaranteeing that the values from two different proposers won't be accepted.

In Raft (see "Raft" on page 300), a leader can discover that its term is out-of-date, which implies the presence of a different leader in the system, and update its term to the more recent one.

In both cases, having a leader is a way to ensure *liveness* (if the current leader has failed, we need a new one), and processes should not take indefinitely long to understand whether or not it has really failed. Lack of *safety* and allowing multiple leaders is a performance optimization: algorithms can proceed with a replication phase, and *safety* is guaranteed by detecting and resolving the conflicts.

We discuss consensus and leader election in the context of consensus in more detail in Chapter 14.

Further Reading

If you'd like to learn more about the concepts mentioned in this chapter, you can refer to the following sources:

Leader election algorithms

Lynch, Nancy and Boaz Patt-Shamir. 1993. "Distributed algorithms." *Lecture notes for 6.852*. Cambridge, MA: MIT.

Attiya, Hagit and Jennifer Welch. 2004. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. USA: John Wiley & Sons.

Tanenbaum, Andrew S. and Maarten van Steen. 2006. *Distributed Systems: Principles and Paradigms* (2nd Ed.). Upper Saddle River, NJ: Prentice-Hall.

CHAPTER 11
Replication and Consistency

Before we move on to discuss consensus and atomic commitment algorithms, let's put together the last piece required for their in-depth understanding: *consistency models*. Consistency models are important, since they explain visibility semantics and behavior of the system in the presence of multiple copies of data.

Fault tolerance is a property of a system that can continue operating correctly in the presence of failures of its components. Making a system fault-tolerant is not an easy task, and it may be difficult to add fault tolerance to the existing system. The primary goal is to remove a single point of failure from the system and make sure that we have redundancy in mission-critical components. Usually, redundancy is entirely transparent for the user.

A system can continue operating correctly by storing multiple copies of data so that, when one of the machines fails, the other one can serve as a failover. In systems with a single source of truth (for example, primary/replica databases), failover can be done explicitly, by promoting a replica to become a new master. Other systems do not require explicit reconfiguration and ensure consistency by collecting responses from multiple participants during read and write queries.

Data *replication* is a way of introducing redundancy by maintaining multiple copies of data in the system. However, since updating multiple copies of data atomically is a problem equivalent to consensus [MILOSEVIC11], it might be quite costly to perform this operation for *every* operation in the database. We can explore some more cost-effective and flexible ways to make data *look* consistent from the user's perspective, while allowing some degree of divergence between participants.

Replication is particularly important in multidatacenter deployments. Geo-replication, in this case, serves multiple purposes: it increases availability and the ability to withstand a failure of one or more datacenters by providing redundancy. It

can also help to reduce the latency by placing a copy of data physically closer to the client.

When data records are modified, their copies have to be updated accordingly. When talking about replication, we care most about three events: *write*, *replica update*, and *read*. These operations trigger a sequence of events initiated by the client. In some cases, updating replicas can happen after the write has finished from the client perspective, but this still does not change the fact that the client has to be able to observe operations in a particular order.

Achieving Availability

We've talked about the fallacies of distributed systems and have identified many things that can go wrong. In the real world, nodes aren't always alive or able to communicate with one another. However, intermittent failures should not impact *availability*: from the user's perspective, the system as a whole has to continue operating as if nothing has happened.

System availability is an incredibly important property: in software engineering, we always strive for high availability, and try to minimize downtime. Engineering teams brag about their uptime metrics. We care so much about availability for several reasons: software has become an integral part of our society, and many important things cannot happen without it: bank transactions, communication, travel, and so on.

For companies, lack of availability can mean losing customers or money: you can't shop in the online store if it's down, or transfer the money if your bank's website isn't responding.

To make the system highly available, we need to design it in a way that allows handling failures or unavailability of one or more participants gracefully. For that, we need to introduce redundancy and replication. However, as soon as we add redundancy, we face the problem of keeping several copies of data in sync and have to implement recovery mechanisms.

Infamous CAP

Availability is a property that measures the ability of the system to serve a response for every request successfully. The theoretical definition of availability mentions eventual response, but of course, in a real-world system, we'd like to avoid services that take indefinitely long to respond.

Ideally, we'd like every operation to be *consistent*. Consistency is defined here as atomic or *linearizable* consistency (see "Linearizability" on page 223). Linearizable history can be expressed as a sequence of instantaneous operations that preserves the original operation order. Linearizability simplifies reasoning about the possible

system states and makes a distributed system appear as if it was running on a single machine.

We would like to achieve both consistency and availability while tolerating network partitions. The network can get split into several parts where processes are not able to communicate with each other: some of the messages sent between partitioned nodes won't reach their destinations.

Availability requires any nonfailing node to deliver results, while consistency requires results to be linearizable. CAP conjecture, formulated by Eric Brewer, discusses trade-offs between Consistency, Availability, and Partition tolerance [BREWER00].

Availability requirement is impossible to satisfy in an asynchronous system, and we cannot implement a system that simultaneously guarantees both *availability* and *consistency* in the presence of *network partitions* [GILBERT02]. We can build systems that guarantee strong consistency while providing *best effort* availability, or guarantee availability while providing *best effort* consistency [GILBERT12]. Best effort here implies that if everything works, the system will not *purposefully* violate any guarantees, but guarantees are allowed to be weakened and violated in the case of network partitions.

In other words, CAP describes a continuum of potential choices, where on different sides of the spectrum we have systems that are:

Consistent and partition tolerant
: CP systems prefer failing requests to serving potentially inconsistent data.

Available and partition tolerant
: AP systems loosen the consistency requirement and allow serving potentially inconsistent values during the request.

An example of a CP system is an implementation of a consensus algorithm, requiring a majority of nodes for progress: always consistent, but might be unavailable in the case of a network partition. A database always accepting writes and serving reads as long as even a single replica is up is an example of an AP system, which may end up losing data or serving inconsistent results.

PACELEC conjecture [ABADI12], an extension of CAP, states that in presence of network partitions there's a choice between consistency and availability (PAC). Else (E), even if the system is running normally, we *still* have to make a choice between latency and consistency.

Use CAP Carefully

It's important to note that CAP discusses *network partitions* rather than *node crashes* or any other type of failure (such as crash-recovery). A node, partitioned from the rest of the cluster, can serve inconsistent requests, but a crashed node will not

respond at all. On the one hand, this implies that it's not necessary to have any nodes down to face consistency problems. On the other hand, this isn't the case in the real world: there are many different failure scenarios (some of which can be simulated with network partitions).

CAP implies that we can face consistency problems even if all the nodes are up, but there are connectivity issues between them since we expect every nonfailed node to respond correctly, with no regard to how many nodes may be down.

CAP conjecture is sometimes illustrated as a triangle, as if we could turn a knob and have more or less of all of the three parameters. However, while we can turn a knob and trade consistency for availability, partition tolerance is a property we cannot realistically tune or trade for anything [HALE10].

Consistency in CAP is defined quite differently from what ACID (see Chapter 5) defines as consistency. ACID consistency describes transaction consistency: transaction brings the database from one valid state to another, maintaining all the database invariants (such as uniqueness constraints and referential integrity). In CAP, it means that operations are *atomic* (operations succeed or fail in their entirety) and *consistent* (operations never leave the data in an inconsistent state).

Availability in CAP is also different from the aforementioned *high availability* [KLEPPMANN15]. The CAP definition puts no bounds on execution latency. Additionally, availability in databases, contrary to CAP, doesn't require *every* nonfailed node to respond to *every* request.

CAP conjecture is used to explain distributed systems, reason about failure scenarios, and evaluate possible situations, but it's important to remember that there's a fine line between *giving up* consistency and serving unpredictable results.

Databases that claim to be on the availability side, when used correctly, are still able to serve consistent results from replicas, given there are enough replicas alive. Of course, there are more complicated failure scenarios and CAP conjecture is just a rule of thumb, and it doesn't necessarily tell the whole truth.[1]

Harvest and Yield

CAP conjecture discusses consistency and availability only in their strongest forms: *linearizability* and the ability of the system to eventually respond to every request.

1 Quorum reads and writes in the context of eventually consistent stores, which are discussed in more detail in "Eventual Consistency" on page 234.

This forces us to make a hard trade-off between the two properties. However, some applications can benefit from slightly relaxed assumptions and we can think about these properties in their weaker forms.

Instead of being *either* consistent *or* available, systems can provide relaxed guarantees. We can define two tunable metrics: *harvest* and *yield*, choosing between which still constitutes correct behavior [FOX99]:

Harvest
: Defines how complete the query is: if the query has to return 100 rows, but can fetch only 99 due to unavailability of some nodes, it still can be better than failing the query completely and returning nothing.

Yield
: Specifies the number of requests that were completed successfully, compared to the total number of attempted requests. Yield is different from the uptime, since, for example, a busy node is not down, but still can fail to respond to some of the requests.

This shifts the focus of the trade-off from the absolute to the relative terms. We can trade harvest for yield and allow some requests to return incomplete data. One of the ways to increase yield is to return query results only from the available partitions (see "Database Partitioning" on page 270). For example, if a subset of nodes storing records of some users is down, we can still continue serving requests for other users. Alternatively, we can require the critical application data to be returned only in its entirety, but allow some deviations for other requests.

Defining, measuring, and making a conscious choice between harvest and yield helps us to build systems that are more resilient to failures.

Shared Memory

For a client, the distributed system storing the data acts as if it has shared storage, similar to a single-node system. Internode communication and message passing are abstracted away and happen behind the scenes. This creates an illusion of a shared memory.

A single unit of storage, accessible by read or write operations, is usually called a *register*. We can view *shared memory* in a distributed database as an array of such registers.

We identify every operation by its *invocation* and *completion* events. We define an operation as *failed* if the process that invoked it crashes before it completes. If both invocation and completion events for one operation happen before the other operation is invoked, we say that this operation *precedes* the other one, and these two operations are *sequential*. Otherwise, we say that they are *concurrent*.

In Figure 11-1, you can see processes P_1 and P_2 executing different operations:

- a) The operation performed by process P_2 starts *after* the operation executed by P_1 has already finished, and the two operations are *sequential*.
- b) There's an overlap between the two operations, so these operations are *concurrent*.
- c) The operation executed by P_2 starts *after* and completes *before* the operation executed by P_1. These operations are *concurrent*, too.

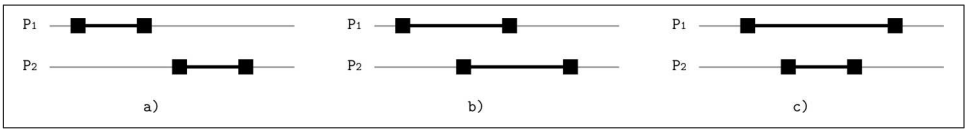

Figure 11-1. Sequential and concurrent operations

Multiple readers or writers can access the register simultaneously. Read and write operations on registers are *not immediate* and take some time. Concurrent read/write operations performed by different processes are not *serial*: depending on how registers behave when operations overlap, they might be ordered differently and may produce different results. Depending on how the register behaves in the presence of concurrent operations, we distinguish among three types of registers:

Safe
: Reads to the safe registers may return *arbitrary* values within the range of the register during a concurrent write operation (which does not sound very practical, but might describe the semantics of an asynchronous system that does not impose the order). Safe registers with binary values might appear to be *flickering* (i.e., returning results alternating between the two values) during reads concurrent to writes.

Regular
: For regular registers, we have slightly stronger guarantees: a read operation can return only the value written by the most recent *completed* write or the value written by the write operation that overlaps with the current read. In this case, the system has some notion of order, but write results are not visible to all the readers simultaneously (for example, this may happen in a replicated database, where the master accepts writes and replicates them to workers serving reads).

Atomic
: Atomic registers guarantee linearizability: every write operation has a single moment before which every read operation returns an old value and after which every read operation returns a new one. Atomicity is a fundamental property that simplifies reasoning about the system state.

Ordering

When we see a sequence of events, we have some intuition about their execution order. However, in a distributed system it's not always that easy, because it's hard to know when *exactly* something has happened and have this information available instantly across the cluster. Each participant may have its view of the state, so we have to look at every operation and define it in terms of its *invocation* and *completion* events and describe the operation bounds.

Let's define a system in which processes can execute `read(register)` and `write(register, value)` operations on shared registers. Each process executes its own set of operations sequentially (i.e., every invoked operation has to complete before it can start the next one). The combination of sequential process executions forms a global history, in which operations can be executed concurrently.

The simplest way to think about consistency models is in terms of read and write operations and ways they can overlap: read operations have no side effects, while writes change the register state. This helps to reason about when exactly data becomes readable after the write. For example, consider a history in which two processes execute the following events concurrently:

```
Process 1:      Process 2:
write(x, 1)     read(x)
                read(x)
```

When looking at these events, it's unclear what is an outcome of the `read(x)` operations in both cases. We have several possible histories:

- Write completes before both reads.
- Write and two reads can get interleaved, and can be executed between the reads.
- Both reads complete before the write.

There's no simple answer to what should happen if we have just one copy of data. In a replicated system, we have more combinations of possible states, and it can get even more complicated when we have multiple processes reading and writing the data.

If all of these operations were executed by the single process, we could enforce a strict order of events, but it's harder to do so with multiple processes. We can group the potential difficulties into two groups:

- Operations may overlap.
- Effects of the nonoverlapping calls might not be visible immediately.

To reason about the operation order and have nonambiguous descriptions of possible outcomes, we have to define consistency models. We discuss concurrency in

distributed systems in terms of shared memory and concurrent systems, since most of the definitions and rules defining consistency still apply. Even though a lot of terminology between concurrent and distributed systems overlap, we can't directly apply most of the concurrent algorithms, because of differences in communication patterns, performance, and reliability.

Consistency Models

Since operations on shared memory registers are allowed to overlap, we should define clear semantics: what happens if multiple clients read or modify different copies of data simultaneously or within a short period. There's no single right answer to that question, since these semantics are different depending on the application, but they are well studied in the context of consistency models.

Consistency models provide different semantics and guarantees. You can think of a consistency model as a contract between the participants: what each replica has to do to satisfy the required semantics, and what users can expect when issuing read and write operations.

Consistency models describe what expectations clients might have in terms of possible returned values despite the existence of multiple copies of data and concurrent accesses to it. In this section, we will discuss *single-operation* consistency models.

Each model describes how far the behavior of the system is from the behavior we might expect or find natural. It helps us to distinguish between "all possible histories" of interleaving operations and "histories permissible under model X," which significantly simplifies reasoning about the visibility of state changes.

We can think about consistency from the perspective of *state*, describe which state invariants are acceptable, and establish allowable relationships between copies of the data placed onto different replicas. Alternatively, we can consider *operation* consistency, which provides an outside view on the data store, describes operations, and puts constraints on the order in which they occur [TANENBAUM06] [AGUILERA16].

Without a global clock, it is difficult to give distributed operations a precise and deterministic order. It's like a Special Relativity Theory for data: every participant has its own perspective on state and time.

Theoretically, we could grab a system-wide lock every time we want to change the system state, but it'd be highly impractical. Instead, we use a set of rules, definitions, and restrictions that limit the number of possible histories and outcomes.

Consistency models add another dimension to what we discussed in "Infamous CAP" on page 216. Now we have to juggle not only consistency and availability, but also consider consistency in terms of synchronization costs [ATTIYA94]. Synchronization

costs may include latency, additional CPU cycles spent executing additional operations, disk I/O used to persist recovery information, wait time, network I/O, and everything else that can be prevented by avoiding synchronization.

First, we'll focus on visibility and propagation of operation results. Coming back to the example with concurrent reads and writes, we'll be able to limit the number of possible histories by either positioning dependent writes after one another or defining a point at which the new value is propagated.

We discuss consistency models in terms of *processes* (clients) issuing `read` and `write` operations against the database state. Since we discuss consistency in the context of replicated data, we assume that the database can have multiple replicas.

Strict Consistency

Strict consistency is the equivalent of complete replication transparency: any write by any process is instantly available for the subsequent reads by any process. It involves the concept of a global clock and, if there was a `write(x, 1)` at instant t_1, any `read(x)` will return a newly written value `1` at *any* instant $t_2 > t_1$.

Unfortunately, this is just a theoretical model, and it's impossible to implement, as the laws of physics and the way distributed systems work set limits on how fast things may happen [SINHA97].

Linearizability

Linearizability is the strongest single-object, single-operation consistency model. Under this model, effects of the write become visible to all readers exactly once at some point in time between its start and end, and no client can observe state transitions or side effects of partial (i.e., unfinished, still in-flight) or incomplete (i.e., interrupted before completion) write operations [LEE15].

Concurrent operations are represented as one of the possible sequential histories for which visibility properties hold. There is some indeterminism in linearizability, as there may exist more than one way in which the events can be ordered [HERLIHY90].

If two operations overlap, they may take effect in any order. All read operations that occur after write operation completion can observe the effects of this operation. As soon as a single read operation returns a particular value, all reads that come after it return the value *at least* as recent as the one it returns [BAILIS14a].

There is some flexibility in terms of the order in which concurrent events occur in a global history, but they cannot be reordered arbitrarily. Operation results should not become effective before the operation starts as that would require an oracle able to

predict future operations. At the same time, results have to take effect before completion, since otherwise, we cannot define a linearization point.

Linearizability respects both sequential process-local operation order and the order of operations running in parallel relative to other processes, and defines a *total order* of the events.

This order should be *consistent*, which means that every read of the shared value should return the latest value written to this shared variable preceding this read, or the value of a write that overlaps with this read. Linearizable write access to a shared variable also implies mutual exclusion: between the two concurrent writes, only one can go first.

Even though operations are concurrent and have some overlap, their effects become visible in a way that makes them appear sequential. No operation happens instantaneously, but still *appears* to be atomic.

Let's consider the following history:

```
Process 1:      Process 2:     Process 3:
write(x, 1)     write(x, 2)    read(x)
                               read(x)
                               read(x)
```

In Figure 11-2, we have three processes, two of which perform write operations on the register `x`, which has an initial value of ∅. Read operations can observe these writes in one of the following ways:

- a) The first read operation can return `1`, `2`, or ∅ (the initial value, a state before both writes), since both writes are still in-flight. The first read can get ordered *before* both writes, *between* the first and second writes, and *after* both writes.
- b) The second read operation can return only `1` and `2`, since the first write has completed, but the second write didn't return yet.
- c) The third read can only return `2`, since the second write is ordered after the first.

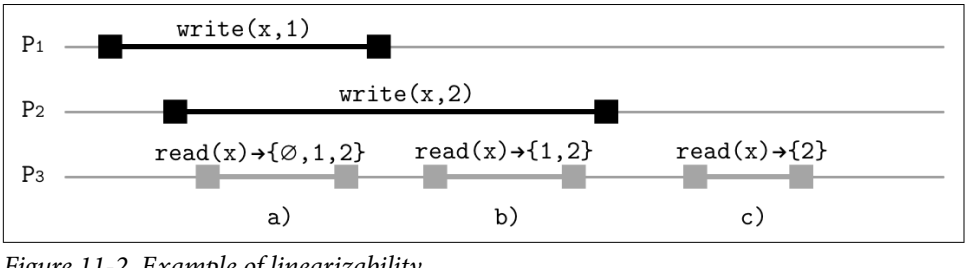

Figure 11-2. Example of linearizability

Linearization point

One of the most important traits of linearizability is visibility: once the operation is complete, everyone must see it, and the system can't "travel back in time," reverting it or making it invisible for some participants. In other words, linearization prohibits stale reads and requires reads to be monotonic.

This consistency model is best explained in terms of atomic (i.e., uninterruptible, indivisible) operations. Operations do not have to *be* instantaneous (also because there's no such thing), but their *effects* have to become visible at some point in time, making an illusion that they were instantaneous. This moment is called a *linearization point*.

Past the linearization point of the write operation (in other words, when the value becomes visible for other processes) every process has to see either the value this operation wrote or some later value, if some additional write operations are ordered after it. A visible value should remain stable until the next one becomes visible after it, and the register should not alternate between the two recent states.

Most of the programming languages these days offer atomic primitives that allow atomic `write` and `compare-and-swap` (CAS) operations. Atomic `write` operations do not consider current register values, unlike CAS, that move from one value to the next only when the previous value is unchanged [HERLIHY94]. Reading the value, modifying it, and then writing it with CAS is more complex than simply checking and setting the value, because of the possible *ABA problem* [DECHEV10]: if CAS expects the value `A` to be present in the register, it will be installed even if the value `B` was set and then switched back to `A` by the other two concurrent write operations. In other words, the presence of the value `A` alone does not guarantee that the value hasn't been changed since the last read.

The linearization point serves as a cutoff, after which operation effects become visible. We can implement it by using locks to guard a critical section, atomic read/write, or read-modify-write primitives.

Figure 11-3 shows that linearizability assumes hard time bounds and the clock is *real time*, so the operation effects have to become visible *between* t_1, when the operation request was issued, and t_2, when the process received a response.

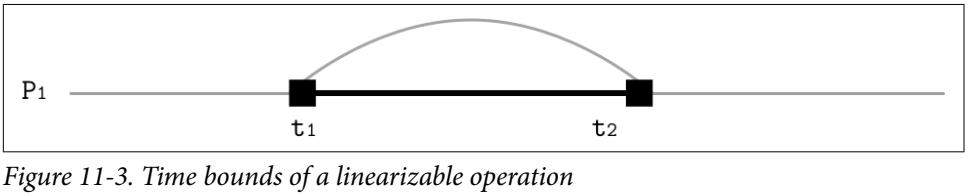

Figure 11-3. Time bounds of a linearizable operation

Figure 11-4 illustrates that the linearization point *cuts* the history into *before* and *after*. Before the linearization point, the old value is visible, after it, the new value is visible.

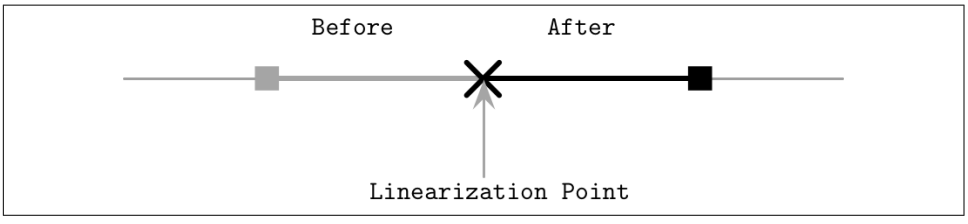

Figure 11-4. Linearization point

Cost of linearizability

Many systems avoid implementing linearizability today. Even CPUs do not offer linearizability when accessing main memory by default. This has happened because synchronization instructions are expensive, slow, and involve cross-node CPU traffic and cache invalidations. However, it is possible to implement linearizability using low-level primitives [MCKENNEY05a], [MCKENNEY05b].

In concurrent programming, you can use compare-and-swap operations to introduce linearizability. Many algorithms work by *preparing* results and then using CAS for swapping pointers and *publishing* them. For example, we can implement a concurrent queue by creating a linked list node and then atomically appending it to the tail of the list [KHANCHANDANI18].

In distributed systems, linearizability requires coordination and ordering. It can be implemented using *consensus*: clients interact with a replicated store using messages, and the consensus module is responsible for ensuring that applied operations are consistent and identical across the cluster. Each write operation will appear instantaneously, exactly once at some point between its invocation and completion events [HOWARD14].

Interestingly, linearizability in its traditional understanding is regarded as a *local* property and implies composition of independently implemented and verified elements. Combining linearizable histories produces a history that is also linearizable [HERLIHY90]. In other words, a system in which all objects are linearizable, is also linearizable. This is a very useful property, but we should remember that its scope is limited to a single object and, even though operations on two independent objects are linearizable, operations that involve both objects have to rely on additional synchronization means.

Reusable Infrastructure for Linearizability

Reusable Infrastructure for Linearizability (RIFL), is a mechanism for implementing linearizable remote procedure calls (RPCs) [LEE15]. In RIFL, messages are uniquely identified with the client ID and a client-local monotonically increasing sequence number.

To assign client IDs, RIFL uses *leases*, issued by the system-wide service: unique identifiers used to establish uniqueness and break sequence number ties. If the failed client tries to execute an operation using an expired lease, its operation will not be committed: the client has to receive a new lease and retry.

If the server crashes before it can acknowledge the write, the client may attempt to retry this operation without knowing that it has already been applied. We can even end up in a situation in which client `C1` writes value `V1`, but doesn't receive an acknowledgment. Meanwhile, client `C2` writes value `V2`. If `C1` retries its operation and successfully writes `V1`, the write of `C2` would be lost. To avoid this, the system needs to prevent repeated execution of retried operations. When the client retries the operation, instead of reapplying it, RIFL returns a completion object, indicating that the operation it's associated with has already been executed, and returns its result.

Completion objects are stored in a durable storage, along with the actual data records. However, their lifetimes are different: the completion object should exist until either the issuing client promises it won't retry the operation associated with it, or until the server detects a client crash, in which case all completion objects associated with it can be safely removed. Creating a completion object should be atomic with the mutation of the data record it is associated with.

Clients have to periodically renew their leases to signal their liveness. If the client fails to renew its lease, it is marked as crashed and all the data associated with its lease is garbage collected. Leases have a limited lifetime to make sure that operations that belong to the failed process won't be retained in the log forever. If the failed client tries to continue operation using an expired lease, its results will not be committed and the client will have to start from scratch.

The advantage of RIFL is that, by guaranteeing that the RPC cannot be executed more than once, an operation can be made linearizable by ensuring that its results are made visible atomically, and most of its implementation details are independent from the underlying storage system.

Sequential Consistency

Achieving linearizability might be too expensive, but it is possible to relax the model, while still providing rather strong consistency guarantees. *Sequential consistency* allows ordering operations as if they were executed in *some* sequential order, while

requiring operations of each individual process to be executed in the same order they were performed by the process.

Processes can observe operations executed by other participants in the order consistent with their own history, but this view can be arbitrarily stale from the global perspective [KINGSBURY18a]. Order of execution *between* processes is undefined, as there's no shared notion of time.

Sequential consistency was initially introduced in the context of concurrency, describing it as a way to execute multiprocessor programs correctly. The original description required memory requests to the same cell to be ordered in the queue (FIFO, arrival order), did not impose global ordering on the overlapping writes to independent memory cells, and allowed reads to fetch the value from the memory cell, or the latest value from the queue if the queue was nonempty [LAMPORT79]. This example helps to understand the semantics of sequential consistency. Operations can be ordered in different ways (depending on the arrival order, or even arbitrarily in case two writes arrive simultaneously), but all processes *observe* the operations in the same order.

Each process can issue read and write requests in an order specified by its own program, which is very intuitive. Any nonconcurrent, single-threaded program executes its steps this way: one after another. All write operations propagating from the same process appear in the order they were submitted by this process. Operations propagating from different sources may be ordered *arbitrarily*, but this order will be consistent from the readers' perspective.

Sequential consistency is often confused with linearizability since both have similar semantics. Sequential consistency, just as linearizability, requires operations to be globally ordered, but linearizability requires the local order of each process and global order to be consistent. In other words, linearizability respects a real-time operation order. Under sequential consistency, ordering holds only for the same-origin writes [VIOTTI16]. Another important distinction is composition: we can combine linearizable histories and still expect results to be linearizable, while sequentially consistent schedules are not composable [ATTIYA94].

Figure 11-5 shows how `write(x,1)` and `write(x,2)` can become visible to P_3 and P_4. Even though in wall-clock terms, `1` was written *before* `2`, it can get ordered after `2`. At the same time, while P_3 already reads the value `1`, P_4 can still read `2`. However, *both* orders, `1 → 2` and `2 → 1`, are valid, as long as they're consistent for different readers. What's important here is that both P_3 and P_4 have observed values *in the same order*: first `2`, and then `1` [TANENBAUM14].

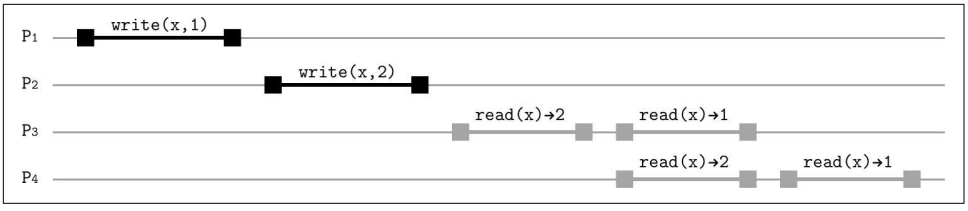

Figure 11-5. Ordering in sequential consistency

Stale reads can be explained, for example, by replica divergence: even though writes propagate to different replicas in the same order, they can arrive there at different times.

The main difference with linearizability is the absence of globally enforced time bounds. Under linearizability, an operation has to become effective within its wall-clock time bounds. By the time the write `W₁` operation completes, its results have to be applied, and every reader should be able to see the value *at least* as recent as one written by `W₁`. Similarly, after a read operation `R₁` returns, any read operation that happens after it should return the value that `R₁` has seen or a later value (which, of course, has to follow the same rule).

Sequential consistency relaxes this requirement: an operation's results can become visible *after* its completion, as long as the order is consistent from the individual processors' perspective. Same-origin writes can't "jump" over each other: their program order, relative to their own executing process, has to be preserved. The other restriction is that the order in which operations have appeared must be consistent for *all* readers.

Similar to linearizability, modern CPUs do not guarantee sequential consistency by default and, since the processor can reorder instructions, we should use memory barriers (also called fences) to make sure that writes become visible to concurrently running threads in order [DREPPER07] [GEORGOPOULOS16].

Causal Consistency

> You see, there is only one constant, one universal, it is the only real truth: causality. Action. Reaction. Cause and effect.
>
> —Merovingian from *The Matrix Reloaded*

Even though having a global operation order is often unnecessary, it might be necessary to establish order between *some* operations. Under the *causal consistency* model, all processes have to see *causally related* operations in the same order. *Concurrent writes* with no causal relationship can be observed in a different order by different processors.

First, let's take a look at *why* we need causality and how writes that have no causal relationship can propagate. In Figure 11-6, processes P_1 and P_2 make writes that *aren't* causally ordered. The results of these operations can propagate to readers at different times and out of order. Process P_3 will see the value `1` before it sees `2`, while P_4 will first see `2`, and then `1`.

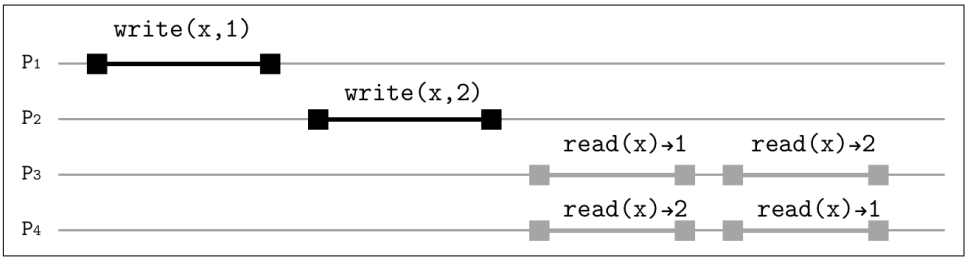

Figure 11-6. Write operations with no causal relationship

Figure 11-7 shows an example of causally related writes. In addition to a written value, we now have to specify a logical clock value that would establish a causal order between operations. P_1 starts with a write operation `write(x,∅,1)`→t_1, which starts from the initial value ∅. P_2 performs another write operation, `write(x, t1, 2)`, and specifies that it is logically ordered *after* t_1, requiring operations to propagate *only* in the order established by the logical clock.

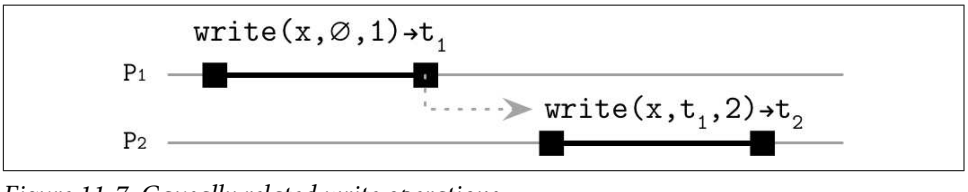

Figure 11-7. Causally related write operations

This establishes a *causal order* between these operations. Even if the latter write propagates faster than the former one, it isn't made visible until all of its dependencies arrive, and the event order is reconstructed from their logical timestamps. In other words, a happened-before relationship is established logically, without using physical clocks, and all processes agree on this order.

Figure 11-8 shows processes P_1 and P_2 making causally related writes, which propagate to P_3 and P_4 in their logical order. This prevents us from the situation shown in Figure 11-6; you can compare histories of P_3 and P_4 in both figures.

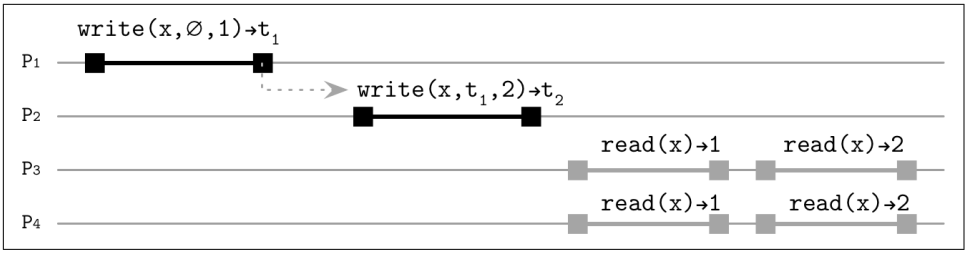

Figure 11-8. Write operations with causal relationship

You can think of this in terms of communication on some online forum: you post something online, someone sees your post and responds to it, and a third person sees this response and continues the conversation thread. It is possible for conversation threads to diverge: you can choose to respond to one of the conversations in the thread and continue the chain of events, but some threads will have only a few messages in common, so there might be no single history for all the messages.

In a causally consistent system, we get session guarantees for the application, ensuring the view of the database is consistent with its own actions, even if it executes read and write requests against different, potentially inconsistent, servers [TERRY94]. These guarantees are: monotonic reads, monotonic writes, read-your-writes, writes-follow-reads. You can find more information on these session models in "Session Models" on page 233.

Causal consistency can be implemented using logical clocks [LAMPORT78] and sending context metadata with every message, summarizing which operations logically precede the current one. When the update is received from the server, it contains the latest version of the context. Any operation can be processed only if all operations preceding it have already been applied. Messages for which contexts do not match are buffered on the server as it is too early to deliver them.

The two prominent and frequently cited projects implementing causal consistency are Clusters of Order-Preserving Servers (COPS) [LLOYD11] and Eiger [LLOYD13]. Both projects implement causality through a library (implemented as a frontend server that users connect to) and track dependencies to ensure consistency. COPS tracks dependencies through key versions, while Eiger establishes operation order instead (operations in Eiger can depend on operations executed on the other nodes; for example, in the case of multipartition transactions). Both projects do not expose out-of-order operations like eventually consistent stores might do. Instead, they detect and handle conflicts: in COPS, this is done by checking the key order and using application-specific functions, while Eiger implements the last-write-wins rule.

Vector clocks

Establishing causal order allows the system to reconstruct the sequence of events even if messages are delivered out of order, fill the gaps between the messages, and avoid publishing operation results in case some messages are still missing. For example, if messages `{M1(∅, t1), M2(M1, t2), M3(M2, t3)}`, each specifying their dependencies, are causally related and were propagated out of order, the process buffers them until it can collect all operation dependencies and restore their causal order [KINGSBURY18b]. Many databases, for example, Dynamo [DECANDIA07] and Riak [SHEEHY10a], use *vector clocks* [LAMPORT78] [MATTERN88] for establishing causal order.

A *vector clock* is a structure for establishing a *partial order* between the events, detecting and resolving divergence between the event chains. With vector clocks, we can simulate common time, global state, and represent asynchronous events as synchronous ones. Processes maintain vectors of *logical clocks*, with one clock per process. Every clock starts at the initial value and is incremented every time a new event arrives (for example, a write occurs). When receiving clock vectors from other processes, a process updates its local vector to the highest clock values per process from the received vectors (i.e., highest clock values the transmitting node has ever seen).

To use vector clocks for conflict resolution, whenever we make a write to the database, we first check if the value for the written key already exists locally. If the previous value already exists, we append a new version to the version vector and establish the causal relationship between the two writes. Otherwise, we start a new chain of events and initialize the value with a single version.

We were talking about consistency in terms of access to shared memory registers and wall-clock operation ordering, and first mentioned potential replica divergence when talking about sequential consistency. Since only write operations to the same memory location have to be ordered, we cannot end up in a situation where we have a write conflict if values are independent [LAMPORT79].

Since we're looking for a consistency model that would improve availability and performance, we have to allow replicas to diverge not only by serving stale reads but also by accepting potentially conflicting writes, so the system is allowed to create two independent chains of events. Figure 11-9 shows such a divergence: from the perspective of one replica, we see history as `1, 5, 7, 8` and the other one reports `1, 5, 3`. Riak allows users to see and resolve divergent histories [DAILY13].

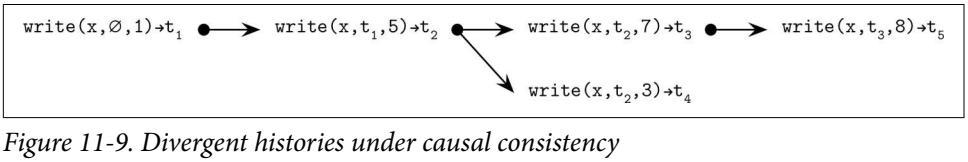

Figure 11-9. Divergent histories under causal consistency

To implement causal consistency, we have to store causal history, add garbage collection, and ask the user to reconcile divergent histories in case of a conflict. Vector clocks can tell you that the conflict has occurred, but do not propose exactly how to resolve it, since resolution semantics are often application-specific. Because of that, some eventually consistent databases, for example, Apache Cassandra, do not order operations causally and use the last-write-wins rule for conflict resolution instead [ELLIS13].

Session Models

Thinking about consistency in terms of value propagation is useful for database developers, since it helps to understand and impose required data invariants, but some things are easier understood and explained from the client point of view. We can look at our distributed system from the perspective of a single client instead of multiple clients.

Session models [VIOTTI16] (also called client-centric consistency models [TANENBAUM06]) help to reason about the state of the distributed system from the client perspective: how each client observes the state of the system while issuing read and write operations.

If other consistency models we discussed so far focus on explaining operation ordering in the presence of concurrent clients, client-centric consistency focuses on how a single client interacts with the system. We still assume that each client's operations are sequential: it has to finish one operation before it can start executing the next one. If the client crashes or loses connection to the server before its operation completes, we do not make any assumptions about the state of incomplete operations.

In a distributed system, clients often can connect to any available replica and, if the results of the recent write against one replica did not propagate to the other one, the client might not be able to observe the state change it has made.

One of the reasonable expectations is that every write issued by the client is visible to it. This assumption holds under the *read-own-writes* consistency model, which states that every read operation following the write on the same or the other replica has to observe the updated value. For example, `read(x)` that was executed immediately after `write(x,V)` will return the value `V`.

The *monotonic reads* model restricts the value visibility and states that if the `read(x)` has observed the value `V`, the following reads have to observe a value at least as recent as `V` or some later value.

The *monotonic writes* model assumes that values originating from the same client appear in the order this client has executed them. If, according to the client session order, `write(x,V2)` was made *after* `write(x,V1)`, their effects have to become visible

in the same order (i.e., `V1` first, and then `V2`) to *all* other processes. Without this assumption, old data can be "resurrected," resulting in data loss.

Writes-follow-reads (sometimes referred as session causality) ensures that writes are ordered after writes that were observed by previous read operations. For example, if `write(x,V2)` is ordered after `read(x)` that has returned `V1`, `write(x,V2)` will be ordered *after* `write(x,V1)`.

Session models make *no* assumptions about operations made by *different* processes (clients) or from the different logical session [TANENBAUM14]. These models describe operation ordering from the point of view of a single process. However, the same guarantees have to hold for *every* process in the system. In other words, if P_1 can read its own writes, P_2 should be able to read *its* own writes, too.

Combining monotonic reads, monotonic writes, and read-own-writes gives Pipelined RAM (PRAM) consistency [LIPTON88] [BRZEZINSKI03], also known as FIFO consistency. PRAM guarantees that write operations originating from one process will propagate in the order they were executed by this process. Unlike under sequential consistency, writes from different processes can be observed in different order.

The properties described by client-centric consistency models are desirable and, in the majority of cases, are used by distributed systems developers to validate their systems and simplify their usage.

Eventual Consistency

Synchronization is expensive, both in multiprocessor programming and in distributed systems. As we discussed in "Consistency Models" on page 222, we can relax consistency guarantees and use models that allow some divergence between the nodes. For example, sequential consistency allows reads to be propagated at different speeds.

Under *eventual consistency*, updates propagate through the system asynchronously. Formally, it states that if there are no *additional* updates performed against the data item, *eventually* all accesses return the latest written value [VOGELS09]. In case of a conflict, the notion of *latest* value might change, as the values from diverged replicas are reconciled using a conflict resolution strategy, such as last-write-wins or using vector clocks (see "Vector clocks" on page 232).

Eventually is an interesting term to describe value propagation, since it specifies no hard time bound in which it has to happen. If the delivery service provides nothing more than an "eventually" guarantee, it doesn't sound like it can be relied upon.

However, in practice, this works well, and many databases these days are described as *eventually consistent*.

Tunable Consistency

Eventually consistent systems are sometimes described in CAP terms: you can trade availability for consistency or vice versa (see "Infamous CAP" on page 216). From the server-side perspective, eventually consistent systems usually implement tunable consistency, where data is replicated, read, and written using three variables:

Replication Factor `N`
: Number of nodes that will store a copy of data.

Write Consistency `W`
: Number of nodes that have to acknowledge a write for it to succeed.

Read Consistency `R`
: Number of nodes that have to respond to a read operation for it to succeed.

Choosing consistency levels where (`R + W > N`), the system can guarantee returning the most recent written value, because there's always an overlap between read and write sets. For example, if `N = 3`, `W = 2`, and `R = 2`, the system can tolerate a failure of just one node. Two nodes out of three must acknowledge the write. In the ideal scenario, the system also asynchronously replicates the write to the third node. If the third node is down, anti-entropy mechanisms (see Chapter 12) eventually propagate it.

During the read, two replicas out of three have to be available to serve the request for us to respond with consistent results. Any combination of nodes will give us at least one node that will have the most up-to-date record for a given key.

When performing a write, the coordinator should submit it to `N` nodes, but can wait for only `W` nodes before it proceeds (or `W - 1` in case the coordinator is also a replica). The rest of the write operations can complete asynchronously or fail. Similarly, when performing a read, the coordinator has to collect *at least* `R` responses. Some databases use speculative execution and submit extra read requests to reduce coordinator response latency. This means if one of the originally submitted read requests fails or arrives slowly, speculative requests can be counted toward `R` instead.

Write-heavy systems may sometimes pick `W = 1` and `R = N`, which allows writes to be acknowledged by just one node before they succeed, but would require *all* the replicas (even potentially failed ones) to be available for reads. The same is true for the `W = N`,

`R = 1` combination: the latest value can be read from any node, as long as writes succeed only after being applied on *all* replicas.

Increasing read or write consistency levels increases latencies and raises requirements for node availability during requests. Decreasing them improves system availability while sacrificing consistency.

Quorums

A consistency level that consists of $\lfloor N/2 \rfloor + 1$ nodes is called a *quorum*, a majority of nodes. In the case of a network partition or node failures, in a system with `2f + 1` nodes, live nodes can continue accepting writes or reads, if up to `f` nodes are unavailable, until the rest of the cluster is available again. In other words, such systems can tolerate at most `f` node failures.

When executing read and write operations using quorums, a system cannot tolerate failures of the majority of nodes. For example, if there are three replicas in total, and two of them are down, read and write operations won't be able to achieve the number of nodes necessary for read and write consistency, since only one node out of three will be able to respond to the request.

Reading and writing using quorums does not guarantee monotonicity in cases of incomplete writes. If some write operation has failed after writing a value to one replica out of three, depending on the contacted replicas, a quorum read can return either the result of the incomplete operation, or the old value. Since subsequent same-value reads are not required to contact the same replicas, values they return can alternate. To achieve read monotonicity (at the cost of availability), we have to use blocking read-repair (see "Read Repair" on page 245).

Witness Replicas

Using quorums for read consistency helps to improve availability: even if some of the nodes are down, a database system can still accept reads and serve writes. The majority requirement guarantees that, since there's an overlap of at least one node in any majority, any quorum read will observe the most recent completed quorum write. However, using replication and majorities increases storage costs: we have to store a copy of the data on each replica. If our replication factor is five, we have to store five copies.

We can improve storage costs by using a concept called *witness replicas*. Instead of storing a copy of the record on each replica, we can split replicas into *copy* and *witness* subsets. Copy replicas still hold data records as previously. Under normal operation, witness replicas merely store the record indicating the fact that the write operation occurred. However, a situation might occur when the number of copy

replicas is too low. For example, if we have three copy replicas and two witness ones, and two copy replicas go down, we end up with a quorum of one copy and two witness replicas.

In cases of write timeouts or copy replica failures, witness replicas can be *upgraded* to temporarily store the record in place of failed or timed-out copy replicas. As soon as the original copy replicas recover, upgraded replicas can revert to their previous state, or recovered replicas can become witnesses.

Let's consider a replicated system with three nodes, two of which are holding copies of data and the third serves as a witness: `[1c, 2c, 3w]`. We attempt to make a write, but `2c` is temporarily unavailable and cannot complete the operation. In this case, we temporarily store the record on the witness replica `3w`. Whenever `2c` comes back up, repair mechanisms can bring it back up-to-date and remove redundant copies from witnesses.

In a different scenario, we can attempt to perform a read, and the record is present on `1c` and `3w`, but not on `2c`. Since any two replicas are enough to constitute a quorum, if any subset of nodes of size two is available, whether it's two copy replicas `[1c, 2c]`, or one copy replica and one witness `[1c, 3w]` or `[2c, 3w]`, we can guarantee to serve consistent results. If we read from `[1c, 2c]`, we fetch the latest record from `1c` and can replicate it to `2c`, since the value is missing there. In case only `[2c, 3w]` are available, the latest record can be fetched from `3w`. To restore the original configuration and bring `2c` up-to-date, the record can be replicated to it, and removed from the witness.

More generally, having `n` copy and `m` witness replicas has same availability guarantees as `n + m` copies, given that we follow two rules:

- Read and write operations are performed using majorities (i.e., with `N/2 + 1` participants)
- At least one of the replicas in this quorum is *necessarily* a copy one

This works because data is guaranteed to be either on the copy or witness replicas. Copy replicas are brought up-to-date by the repair mechanism in case of a failure, and witness replicas store the data in the interim.

Using witness replicas helps to reduce storage costs while preserving consistency invariants. There are several implementations of this approach; for example, Spanner [CORBETT12] and Apache Cassandra.

Strong Eventual Consistency and CRDTs

We've discussed several strong consistency models, such as linearizability and serializability, and a form of weak consistency: eventual consistency. A possible middle ground between the two, offering some benefits of both models, is *strong eventual consistency*. Under this model, updates are allowed to propagate to servers late or out of order, but when all updates finally propagate to target nodes, conflicts between them can be resolved and they can be merged to produce the same valid state [GOMES17].

Under some conditions, we can relax our consistency requirements by allowing operations to preserve additional state that allows the diverged states to be reconciled (in other words, merged) after execution. One of the most prominent examples of such an approach is *Conflict-Free Replicated Data Types* (CRDTs, [SHAPIRO11a]) implemented, for example, in Redis [BIYIKOGLU13].

CRDTs are specialized data structures that preclude the existence of conflict and allow operations on these data types to be applied in any order without changing the result. This property can be extremely useful in a distributed system. For example, in a multinode system that uses conflict-free replicated counters, we can increment counter values on each node independently, even if they cannot communicate with one another due to a network partition. As soon as communication is restored, results from all nodes can be reconciled, and none of the operations applied during the partition will be lost.

This makes CRDTs useful in eventually consistent systems, since replica states in such systems are allowed to temporarily diverge. Replicas can execute operations locally, without prior synchronization with other nodes, and operations eventually propagate to all other replicas, potentially out of order. CRDTs allow us to reconstruct the complete system state from local individual states or operation sequences.

The simplest example of CRDTs is operation-based Commutative Replicated Data Types (CmRDTs). For CmRDTs to work, we need the allowed operations to be:

Side-effect free
: Their application does not change the system state.

Commutative
: Argument order does not matter: `x • y = y • x`. In other words, it doesn't matter whether `x` is merged with `y`, or `y` is merged with `x`.

Causally ordered
: Their successful delivery depends on the precondition, which ensures that the system has reached the state the operation can be applied to.

For example, we could implement a *grow-only counter*. Each server can hold a state vector consisting of last known counter updates from all other participants, initialized with zeros. Each server is only allowed to modify its own value in the vector. When updates are propagated, the function `merge(state1, state2)` merges the states from the two servers.

For example, we have three servers, with initial state vectors initialized:

```
Node 1:          Node 2:          Node 3:
[0, 0, 0]        [0, 0, 0]        [0, 0, 0]
```

If we update counters on the first and third nodes, their states change as follows:

```
Node 1:          Node 2:          Node 3:
[1, 0, 0]        [0, 0, 0]        [0, 0, 1]
```

When updates propagate, we use a merge function to combine the results by picking the maximum value for each slot:

```
Node 1 (Node 3 state vector propagated):
merge([1, 0, 0], [0, 0, 1]) = [1, 0, 1]

Node 2 (Node 1 state vector propagated):
merge([0, 0, 0], [1, 0, 0]) = [1, 0, 0]

Node 2 (Node 3 state vector propagated):
merge([1, 0, 0], [0, 0, 1]) = [1, 0, 1]

Node 3 (Node 1 state vector propagated):
merge([0, 0, 1], [1, 0, 0]) = [1, 0, 1]
```

To determine the current vector state, the sum of values in all slots is computed: `sum([1, 0, 1]) = 2`. The merge function is commutative. Since servers are only allowed to update their own values and these values are independent, no additional coordination is required.

It is possible to produce a *Positive-Negative-Counter* (PN-Counter) that supports both increments and decrements by using payloads consisting of two vectors: `P`, which nodes use for increments, and `N`, where they store decrements. In a larger system, to avoid propagating huge vectors, we can use *super-peers*. Super-peers replicate counter states and help to avoid constant peer-to-peer chatter [SHAPIRO11b].

To save and replicate values, we can use *registers*. The simplest version of the register is the *last-write-wins* register (LWW register), which stores a unique, globally ordered timestamp attached to each value to resolve conflicts. In case of a conflicting write, we preserve only the one with the larger timestamp. The merge operation (picking the value with the largest timestamp) here is also commutative, since it relies on the timestamp. If we cannot allow values to be discarded, we can supply application-specific merge logic and use a *multivalue* register, which stores all values that were written and allows the application to pick the right one.

Another example of CRDTs is an unordered *grow-only* set (G-Set). Each node maintains its local state and can append elements to it. Adding elements produces a valid set. Merging two sets is also a commutative operation. Similar to counters, we can use two sets to support both additions and removals. In this case, we have to preserve an invariant: only the values contained in the addition set can be added into the removal set. To reconstruct the current state of the set, all elements contained in the removal set are subtracted from the addition set [SHAPIRO11b].

An example of a conflict-free type that combines more complex structures is a conflict-free replicated JSON data type, allowing modifications such as insertions, deletions, and assignments on deeply nested JSON documents with list and map types. This algorithm performs merge operations on the client side and does not require operations to be propagated in any specific order [KLEPPMANN14].

There are quite a few possibilities CRDTs provide us with, and we can see more data stores using this concept to provide Strong Eventual Consistency (SEC). This is a powerful concept that we can add to our arsenal of tools for building fault-tolerant distributed systems.

Summary

Fault-tolerant systems use replication to improve availability: even if some processes fail or are unresponsive, the system as a whole can continue functioning correctly. However, keeping multiple copies in sync requires additional coordination.

We've discussed several single-operation consistency models, ordered from the one with the most guarantees to the one with the least:[2]

Linearizability
: Operations appear to be applied instantaneously, and the real-time operation order is maintained.

Sequential consistency
: Operation effects are propagated in *some* total order, and this order is consistent with the order they were executed by the individual processes.

Causal consistency
: Effects of the causally related operations are visible in the same order to all processes.

2 These short definitions are given for recap only, the reader is advised to refer to the complete definitions for context.

PRAM/FIFO consistency
: Operation effects become visible in the same order they were executed by individual processes. Writes from different processes can be observed in different orders.

After that, we discussed multiple session models:

Read-own-writes
: Read operations reflect the previous writes. Writes propagate through the system and become available for later reads that come from the same client.

Monotonic reads
: Any read that has observed a value cannot observe a value that is older that the observed one.

Monotonic writes
: Writes coming from the same client propagate to other clients in the order they were made by this client.

Writes-follow-reads
: Write operations are ordered after the writes whose effects were observed by the previous reads executed by the same client.

Knowing and understanding these concepts can help you to understand the guarantees of the underlying systems and use them for application development. Consistency models describe rules that operations on data have to follow, but their scope is limited to a specific system. Stacking systems with weaker guarantees on top of ones with stronger guarantees or ignoring consistency implications of underlying systems may lead to unrecoverable inconsistencies and data loss.

We also discussed the concept of *eventual* and *tunable* consistency. Quorum-based systems use majorities to serve consistent data. *Witness replicas* can be used to reduce storage costs.

Further Reading

If you'd like to learn more about the concepts mentioned in this chapter, you can refer to the following sources:

Consistency models

Perrin, Matthieu. 2017. *Distributed Systems: Concurrency and Consistency* (1st Ed.). Elsevier, UK: ISTE Press.

Viotti, Paolo and Marko Vukolić. 2016. "Consistency in Non-Transactional Distributed Storage Systems." *ACM Computing Surveys* 49, no. 1 (July): Article 19. *https://doi.org/0.1145/2926965*.

Bailis, Peter, Aaron Davidson, Alan Fekete, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2013. "Highly available transactions: virtues and limitations." *Proceedings of the VLDB Endowment* 7, no. 3 (November): 181-192. *https://doi.org/10.14778/2732232.2732237*.

Aguilera, M.K., and D.B. Terry. 2016. "The Many Faces of Consistency." *Bulletin of the Technical Committee on Data Engineering* 39, no. 1 (March): 3-13.

CHAPTER 12
Anti-Entropy and Dissemination

Most of the communication patterns we've been discussing so far were either peer-to-peer or one-to-many (coordinator and replicas). To reliably propagate data records throughout the system, we need the propagating node to be available and able to reach the other nodes, but even then the throughput is limited to a single machine.

Quick and reliable propagation may be less applicable to data records and more important for the cluster-wide metadata, such as membership information (joining and leaving nodes), node states, failures, schema changes, etc. Messages containing this information are generally infrequent and small, but have to be propagated as quickly and reliably as possible.

Such updates can generally be propagated to all nodes in the cluster using one of the three broad groups of approaches [DEMERS87]; schematic depictions of these communication patterns are shown in Figure 12-1:

- a) Notification broadcast from one process to *all* others.
- b) Periodic peer-to-peer information exchange. Peers connect pairwise and exchange messages.
- c) Cooperative broadcast, where message recipients become broadcasters and help to spread the information quicker and more reliably.

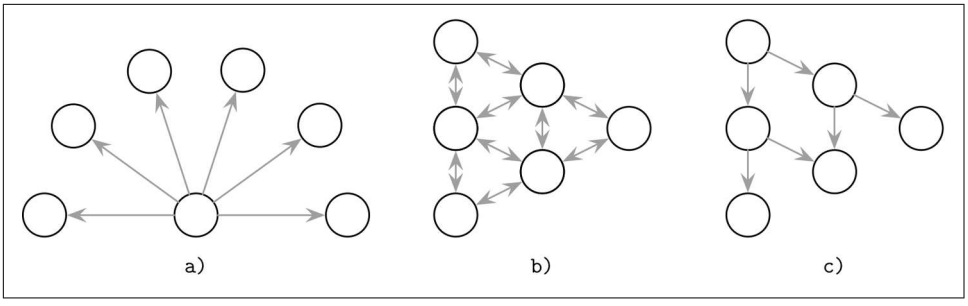

Figure 12-1. Broadcast (a), anti-entropy (b), and gossip (c)

Broadcasting the message to all other processes is the most straightforward approach that works well when the number of nodes in the cluster is small, but in large clusters it can get *expensive* because of the number of nodes, and *unreliable* because of overdependence on a single process. Individual processes may not always know about the existence of all other processes in the network. Moreover, there has to be some overlap in time during which both the broadcasting process and *each one* of its recipients are up, which might be difficult to achieve in some cases.

To relax these constraints, we can assume that *some* updates may fail to propagate. The coordinator will do its best and deliver the messages to all available participants, and then anti-entropy mechanisms will bring nodes back in sync in case there were any failures. This way, the responsibility for delivering messages is shared by all nodes in the system, and is split into two steps: primary delivery and periodic sync.

Entropy is a property that represents the measure of disorder in the system. In a distributed system, entropy represents a degree of state divergence between the nodes. Since this property is undesired and its amount should be kept to a minimum, there are many techniques that help to deal with entropy.

Anti-entropy is usually used to bring the nodes back up-to-date in case the primary delivery mechanism has failed. The system can continue functioning correctly even if the coordinator fails at some point, since the other nodes will continue spreading the information. In other words, anti-entropy is used to lower the convergence time bounds in eventually consistent systems.

To keep nodes in sync, anti-entropy triggers a background or a foreground process that compares and reconciles missing or conflicting records. Background anti-entropy processes use auxiliary structures such as Merkle trees and update logs to identify divergence. Foreground anti-entropy processes piggyback read or write requests: hinted handoff, read repairs, etc.

If replicas diverge in a replicated system, to restore consistency and bring them back in sync, we have to find and repair missing records by comparing replica states pairwise. For large datasets, this can be very costly: we have to read the whole dataset on

both nodes and notify replicas about more recent state changes that weren't yet propagated. To reduce this cost, we can consider ways in which replicas can get out-of-date and patterns in which data is accessed.

Read Repair

It is easiest to detect divergence between the replicas during the read, since at that point we can contact replicas, request the queried state from each one of them, and see whether or not their responses match. Note that in this case we do not query an entire dataset stored on each replica, and we limit our goal to just the data that was requested by the client.

The coordinator performs a distributed read, optimistically assuming that replicas are in sync and have the same information available. If replicas send different responses, the coordinator sends missing updates to the replicas where they're missing.

This mechanism is called *read repair*. It is often used to detect and eliminate inconsistencies. During read repair, the coordinator node makes a request to replicas, waits for their responses, and compares them. In case some of the replicas have missed the recent updates and their responses differ, the coordinator detects inconsistencies and sends updates back to the replicas [DECANDIA07].

Some Dynamo-style databases choose to lift the requirement of contacting *all* replicas and use tunable consistency levels instead. To return consistent results, we do not have to contact and repair all the replicas, but only the number of nodes that satisfies the consistency level. If we do *quorum* reads and writes, we still get consistent results, but some of the replicas still might not contain all the writes.

Read repair can be implemented as a *blocking* or *asynchronous* operation. During blocking read repair, the original client request has to wait until the coordinator "repairs" the replicas. Asynchronous read repair simply schedules a task that can be executed after results are returned to the user.

Blocking read repair ensures read monotonicity (see "Session Models" on page 233) for quorum reads: as soon as the client reads a specific value, subsequent reads return the value at least as recent as the one it has seen, since replica states were repaired. If we're not using quorums for reads, we lose this monotonicity guarantee as data might have not been propagated to the target node by the time of a subsequent read. At the same time, blocking read repair sacrifices availability, since repairs should be acknowledged by the target replicas and the read cannot return until they respond.

To detect exactly which records differ between replica responses, some databases (for example, Apache Cassandra) use specialized iterators with merge listeners, which reconstruct differences between the merged result and individual inputs. Its output is then used by the coordinator to notify replicas about the missing data.

Read repair assumes that replicas are *mostly* in sync and we do not expect every request to fall back to a blocking repair. Because of the read monotonicity of blocking repairs, we can also expect subsequent requests to return the same consistent results, as long as there was no write operation that has completed in the interim.

Digest Reads

Instead of issuing a full read request to each node, the coordinator can issue only one full read request and send only *digest* requests to the other replicas. A digest request reads the replica-local data and, instead of returning a full snapshot of the requested data, it computes a hash of this response. Now, the coordinator can compute a hash of the full read and compare it to digests from all other nodes. If all the digests match, it can be confident that the replicas are in sync.

In case digests do not match, the coordinator does not know which replicas are ahead, and which ones are behind. To bring lagging replicas back in sync with the rest of the nodes, the coordinator has to issue full reads to any replicas that responded with different digests, compare their responses, reconcile the data, and send updates to the lagging replicas.

Digests are usually computed using a noncryptographic hash function, such as MD5, since it has to be computed quickly to make the "happy path" performant. Hash functions can have *collisions*, but their probability is negligible for most real-world systems. Since databases often use more than just one anti-entropy mechanism, we can expect that, even in the unlikely event of a hash collision, data will be reconciled by the different subsystem.

Hinted Handoff

Another anti-entropy approach is called *hinted handoff* [DECANDIA07], a write-side repair mechanism. If the target node fails to acknowledge the write, the write coordinator or one of the replicas stores a special record, called a *hint*, which is replayed to the target node as soon as it comes back up.

In Apache Cassandra, unless the `ANY` consistency level is in use [ELLIS11], hinted writes aren't counted toward the replication factor (see "Tunable Consistency" on page 235), since the data in the hint log isn't accessible for reads and is only used to help the lagging participants catch up.

Some databases, for example Riak, use *sloppy quorums* together with hinted handoff. With sloppy quorums, in case of replica failures, write operations can use additional healthy nodes from the node list, and these nodes do not have to be target replicas for the executed operations.

For example, say we have a five-node cluster with nodes `{A, B, C, D, E}`, where `{A, B, C}` are replicas for the executed write operation, and node `B` is down. `A`, being the coordinator for the query, picks node `D` to satisfy the sloppy quorum and maintain the desired availability and durability guarantees. Now, data is replicated to `{A, D, C}`. However, the record at `D` will have a hint in its metadata, since the write was originally intended for `B`. As soon as `B` recovers, `D` will attempt to forward a hint back to it. Once the hint is replayed on `B`, it can be safely removed without reducing the total number of replicas [DECANDIA07].

Under similar circumstances, if nodes `{B, C}` are briefly separated from the rest of the cluster by the network partition, and a sloppy quorum write was done against `{A, D, E}`, a read on `{B, C}`, immediately following this write, would *not* observe the latest read [DOWNEY12]. In other words, sloppy quorums improve availability at the cost of consistency.

Merkle Trees

Since read repair can only fix inconsistencies on the currently queried data, we should use different mechanisms to find and repair inconsistencies in the data that is not actively queried.

As we already discussed, finding exactly which rows have diverged between the replicas requires exchanging and comparing the data records pairwise. This is highly impractical and expensive. Many databases employ *Merkle trees* [MERKLE87] to reduce the cost of reconciliation.

Merkle trees compose a compact hashed representation of the local data, building a tree of hashes. The lowest level of this hash tree is built by scanning an entire table holding data records, and computing hashes of record ranges. Higher tree levels contain hashes of the lower-level hashes, building a hierarchical representation that allows us to quickly detect inconsistencies by comparing the hashes, following the hash tree nodes recursively to narrow down inconsistent ranges. This can be done by exchanging and comparing subtrees level-wise, or by exchanging and comparing entire trees.

Figure 12-2 shows a composition of a Merkle tree. The lowest level consists of the hashes of data record ranges. Hashes for each higher level are computed by hashing underlying level hashes, repeating this process recursively up to the tree root.

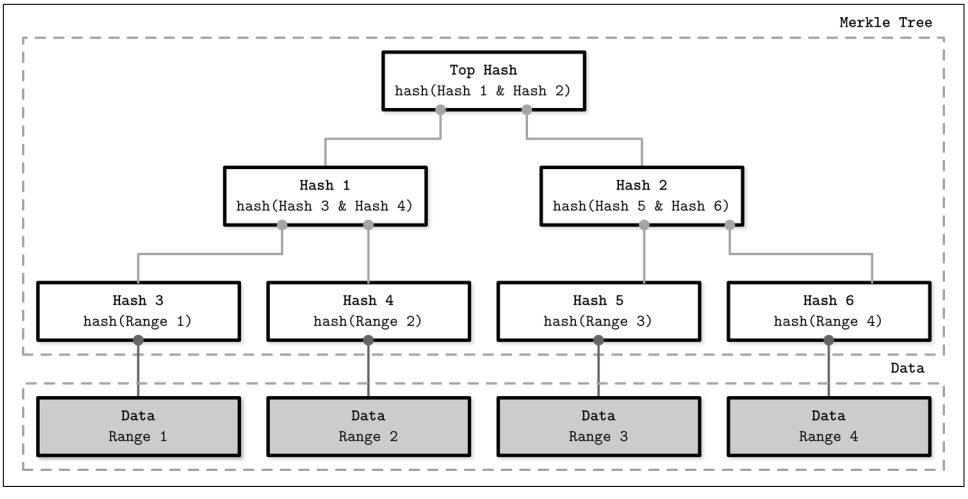

Figure 12-2. Merkle tree. Gray boxes represent data record ranges. White boxes represent a hash tree hierarchy.

To determine whether or not there's an inconsistency between the two replicas, we only need to compare the root-level hashes from their Merkle trees. By comparing hashes pairwise from top to bottom, it is possible to locate ranges holding differences between the nodes, and repair data records contained in them.

Since Merkle trees are calculated recursively from the bottom to the top, a change in data triggers recomputation of the entire subtree. There's also a trade-off between the size of a tree (consequently, sizes of exchanged messages) and its precision (how small and exact data ranges are).

Bitmap Version Vectors

More recent research on this subject introduces *bitmap version vectors* [GONÇALVES15], which can be used to resolve data conflicts based on *recency*: each node keeps a per-peer log of operations that have occurred locally or were replicated. During anti-entropy, logs are compared, and missing data is replicated to the target node.

Each write, coordinated by a node, is represented by a *dot* `(i,n)`: an event with a node-local sequence number `i` coordinated by the node `n`. The sequence number `i` starts with `1` and is incremented each time the node executes a write operation.

To track replica states, we use node-local logical clocks. Each clock represents a set of dots, representing writes this node has seen *directly* (coordinated by the node itself), or *transitively* (coordinated by and replicated from the other nodes).

In the node logical clock, events coordinated by the node itself will have no gaps. If some writes aren't replicated from the other nodes, the clock will contain gaps. To get

two nodes back in sync, they can exchange logical clocks, identify gaps represented by the missing dots, and then replicate data records associated with them. To do this, we need to reconstruct the data records each dot refers to. This information is stored in a *dotted causal container* (DCC), which maps dots to causal information for a given key. This way, conflict resolution captures causal relationships between the writes.

Figure 12-3 (adapted from [GONÇALVES15]) shows an example of the state representation of three nodes in the system, P_1, P_2 and P_3, from the perspective of P_2, tracking which values it has seen. Each time P_2 makes a write or receives a replicated value, it updates this table.

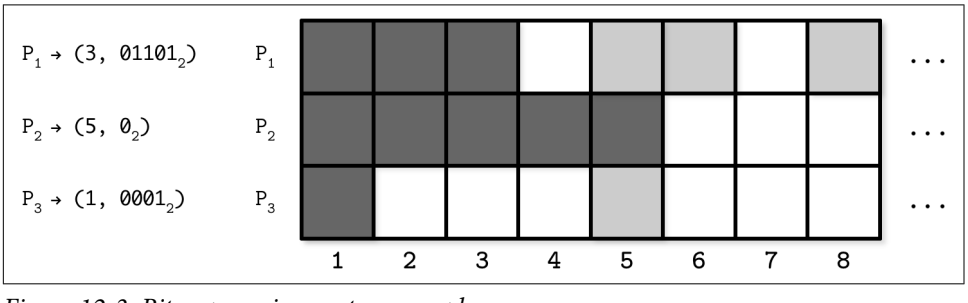

Figure 12-3. Bitmap version vector example

During replication, P_2 creates a compact representation of this state and creates a map from the node identifier to a pair of latest values, up to which it has seen consecutive writes, and a bitmap where other seen writes are encoded as 1. $(3, 01101_2)$ here means that node P_2 has seen consecutive updates up to the third value, and it has seen values on the second, third, and fifth position relative to 3 (i.e., it has seen the values with sequence numbers 5, 6, and 8).

During exchange with other nodes, it will receive the missing updates the other node has seen. As soon as all the nodes in the system have seen consecutive values up to the index `i`, the version vector can be truncated up to this index.

An advantage of this approach is that it captures the causal relation between the value writes and allows nodes to precisely identify the data points missing on the other nodes. A possible downside is that, if the node was down for an extended time period, peer nodes can't truncate the log, since data still has to be replicated to the lagging node once it comes back up.

Gossip Dissemination

> Masses are always breeding grounds of psychic epidemics.
>
> —Carl Jung

To involve other nodes, and propagate updates with the *reach* of a broadcast and the *reliability* of anti-entropy, we can use gossip protocols.

Gossip protocols are probabilistic communication procedures based on how rumors are spread in human society or how diseases propagate in the population. Rumors and epidemics provide rather illustrative ways to describe how these protocols work: rumors spread while the population still has an interest in hearing them; diseases propagate until there are no more susceptible members in the population.

The main objective of gossip protocols is to use cooperative propagation to disseminate information from one process to the rest of the cluster. Just as a virus spreads through the human population by being passed from one individual to another, potentially increasing in scope with each step, information is relayed through the system, getting more processes involved.

A process that holds a record that has to be spread around is said to be *infective*. Any process that hasn't received the update yet is then *susceptible*. Infective processes not willing to propagate the new state after a period of active dissemination are said to be *removed* [DEMERS87]. All processes start in a susceptible state. Whenever an update for some data record arrives, a process that received it moves to the infective state and starts disseminating the update to other *random* neighboring processes, infecting them. As soon as the infective processes become certain that the update was propagated, they move to the removed state.

To avoid explicit coordination and maintaining a global list of recipients and requiring a single coordinator to broadcast messages to each other participant in the system, this class of algorithms models completeness using the *loss of interest* function. The protocol efficiency is then determined by how quickly it can *infect* as many nodes as possible, while keeping overhead caused by redundant messages to a minimum.

Gossip can be used for asynchronous message delivery in homogeneous decentralized systems, where nodes may not have long-term membership or be organized in any topology. Since gossip protocols generally do not require explicit coordination, they can be useful in systems with flexible membership (where nodes are joining and leaving frequently) or mesh networks.

Gossip protocols are very robust and help to achieve high reliability in the presence of failures inherent to distributed systems. Since messages are relayed in a randomized manner, they still can be delivered even if some communication components between them fail, just through the different paths. It can be said that the system adapts to failures.

Gossip Mechanics

Processes periodically select `f` peers at random (where `f` is a configurable parameter, called *fanout*) and exchange currently "hot" information with them. Whenever the process learns about a new piece of information from its peers, it will attempt to pass it on further. Because peers are selected probabilistically, there will always be some overlap, and messages will get delivered repeatedly and may continue circulating for some time. *Message redundancy* is a metric that captures the overhead incurred by repeated delivery. Redundancy is an important property, and it is crucial to how gossip works.

The amount of time the system requires to reach convergence is called *latency*. There's a slight difference between reaching convergence (stopping the gossip process) and delivering the message to all peers, since there might be a short period during which all peers are notified, but gossip continues. Fanout and latency depend on the system size: in a larger system, we either have to increase the fanout to keep latency stable, or allow higher latency.

Over time, as the nodes notice they've been receiving the same information again and again, the message will start losing importance and nodes will have to eventually stop relaying it. Interest loss can be computed either *probabilistically* (the probability of propagation stop is computed for each process on every step) or using a *threshold* (the number of received duplicates is counted, and propagation is stopped when this number is too high). Both approaches have to take the cluster size and fanout into consideration. Counting duplicates to measure convergence can improve latency and reduce redundancy [DEMERS87].

In terms of consistency, gossip protocols offer *convergent* consistency [BIRMAN07]: nodes have a higher probability to have the same view of the events that occurred further in the past.

Overlay Networks

Even though gossip protocols are important and useful, they're usually applied for a narrow set of problems. Nonepidemic approaches can distribute the message with nonprobabilistic certainty, less redundancy, and generally in a more optimal way [BIRMAN07]. Gossip algorithms are often praised for their scalability and the fact it is possible to distribute a message within `log N` message rounds (where `N` is the size of the cluster) [KREMARREC07], but it's important to keep the number of *redundant* messages generated during gossip rounds in mind as well. To achieve reliability, gossip-based protocols produce *some* duplicate message deliveries.

Selecting nodes at random greatly improves system *robustness*: if there is a network partition, messages will be delivered eventually if there are links that indirectly connect two processes. The obvious downside of this approach is that it is not message-

optimal: to guarantee robustness, we have to maintain redundant connections between the peers and send redundant messages.

A middle ground between the two approaches is to construct a *temporary* fixed topology in a gossip system. This can be achieved by creating an *overlay network* of peers: nodes can sample their peers and select the best contact points based on proximity (usually measured by the latency).

Nodes in the system can form *spanning trees*: unidirected, loop-free graphs with distinct edges, covering the whole network. Having such a graph, messages can be distributed in a fixed number of steps.

Figure 12-4 shows an example of a spanning tree:[1]

- a) We achieve full connectivity between the points without using all the edges.
- b) We can lose connectivity to the entire subtree if just a single link is broken.

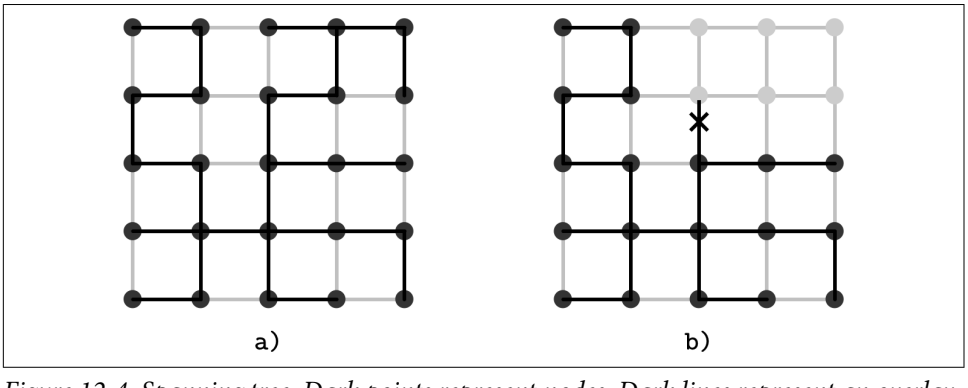

Figure 12-4. Spanning tree. Dark points represent nodes. Dark lines represent an overlay network. Gray lines represent other possible existing connections between the nodes.

One of the potential downsides of this approach is that it might lead to forming interconnected "islands" of peers having strong preferences toward each other.

To keep the number of messages low, while allowing quick recovery in case of a connectivity loss, we can mix both approaches—fixed topologies and tree-based broadcast—when the system is in a *stable* state, and fall back to gossip for *failover* and system recovery.

1 This example is only used for illustration: nodes in the network are generally not arranged in a grid.

Hybrid Gossip

Push/lazy-push multicast trees (Plumtrees) [LEITAO07] make a trade-off between epidemic and tree-based broadcast primitives. Plumtrees work by creating a spanning tree overlay of nodes to *actively* distribute messages with the smallest overhead. Under normal conditions, nodes send full messages to just a small subset of peers provided by the peer sampling service.

Each node sends the full message to the small subset of nodes, and for the rest of the nodes, it *lazily* forwards only the message ID. If the node receives the identifier of a message it has never seen, it can query its peers to get it. This *lazy-push* step ensures high reliability and provides a way to quickly heal the broadcast tree. In case of failures, protocol falls back to the gossip approach through lazy-push steps, broadcasting the message and repairing the overlay.

Due to the nature of distributed systems, any node or link between the nodes might fail at any time, making it impossible to traverse the tree when the segment becomes unreachable. The lazy gossip network helps to notify peers about seen messages in order to construct and repair the tree.

Figure 12-5 shows an illustration of such double connectivity: nodes are connected with an optimal spanning tree (solid lines) and the lazy gossip network (dotted lines). This illustration does not represent any particular network topology, but only *connections* between the nodes.

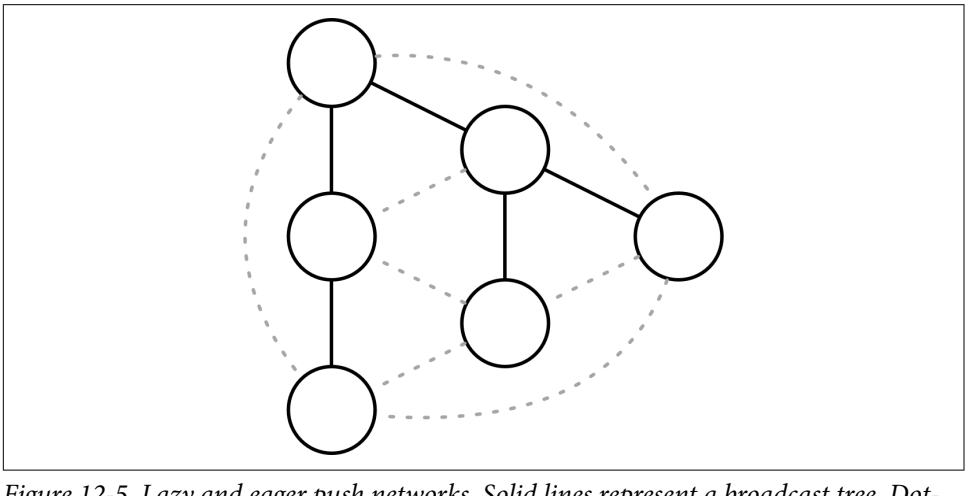

Figure 12-5. Lazy and eager push networks. Solid lines represent a broadcast tree. Dotted lines represent lazy gossip connections.

One of the advantages of using the lazy-push mechanism for tree construction and repair is that in a network with constant load, it will tend to generate a tree that also

minimizes message latency, since nodes that are first to respond are added to the broadcast tree.

Partial Views

Broadcasting messages to all known peers and maintaining a full view of the cluster can get expensive and impractical, especially if the *churn* (measure of the number of joining and leaving nodes in the system) is high. To avoid this, gossip protocols often use a *peer sampling service*. This service maintains a *partial view* of the cluster, which is periodically refreshed using gossip. Partial views overlap, as some degree of redundancy is desired in gossip protocols, but too much redundancy means we're doing extra work.

For example, the Hybrid Partial View (HyParView) protocol [LEITAO07] maintains a small *active* view and a larger *passive* view of the cluster. Nodes from the active view create an overlay that can be used for dissemination. Passive view is used to maintain a list of nodes that can be used to replace the failed ones from the active view.

Periodically, nodes perform a shuffle operation, during which they exchange their active and passive views. During this exchange, nodes add the members from both passive and active views they receive from their peers to their passive views, cycling out the oldest values to cap the list size.

The active view is updated depending on the state changes of nodes in this view and requests from peers. If a process P_1 suspects that P_2, one of the peers from its active view, has failed, P_1 removes P_2 from its active view and attempts to establish a connection with a replacement process P_3 from the passive view. If the connection fails, P_3 is removed from the passive view of P_1.

Depending on the number of processes in P_1's active view, P_3 may choose to decline the connection if its active view is already full. If P_1's view is empty, P_3 *has to* replace one of its current active view peers with P_1. This helps bootstrapping or recovering nodes to quickly become effective members of the cluster at the cost of cycling some connections.

This approach helps to reduce the number of messages in the system by using only active view nodes for dissemination, while maintaining high reliability by using passive views as a recovery mechanism. One of the performance and quality measures is how quickly a peer sampling service converges to a stable overlay in cases of topology reorganization [JELASITY04]. HyParView scores rather high here, because of how the views are maintained and since it gives priority to bootstrapping processes.

HyParView and Plumtree use a *hybrid gossip* approach: using a small subset of peers for broadcasting messages and falling back to a wider network of peers in case of failures and network partitions. Both systems do not rely on a global view that

includes all the peers, which can be helpful not only because of a large number of nodes in the system (which is not the case most of the time), but also because of costs associated with maintaining an up-to-date list of members on every node. Partial views allow nodes to actively communicate with only a small subset of neighboring nodes.

Summary

Eventually consistent systems allow replica state divergence. Tunable consistency allows us to trade consistency for availability and vice versa. Replica divergence can be resolved using one of the anti-entropy mechanisms:

Hinted handoff
: Temporarily store writes on neighboring nodes in case the target is down, and replay them on the target as soon as it comes back up.

Read-repair
: Reconcile requested data ranges during the read by comparing responses, detecting missing records, and sending them to lagging replicas.

Merkle trees
: Detect data ranges that require repair by computing and exchanging hierarchical trees of hashes.

Bitmap version vectors
: Detect missing replica writes by maintaining compact records containing information about the most recent writes.

These anti-entropy approaches optimize for one of the three parameters: scope reduction, recency, or completeness. We can reduce the scope of anti-entropy by only synchronizing the data that is being actively queried (read-repairs) or individual missing writes (hinted handoff). If we assume that most failures are temporary and participants recover from them as quickly as possible, we can store the log of the most recent diverged events and know exactly what to synchronize in the event of failure (bitmap version vectors). If we need to compare entire datasets on multiple nodes pairwise and efficiently locate differences between them, we can hash the data and compare hashes (Merkle trees).

To reliably distribute information in a large-scale system, gossip protocols can be used. Hybrid gossip protocols reduce the number of exchanged messages while remaining resistant to network partitions, when possible.

Many modern systems use gossip for failure detection and membership information [DECANDIA07]. HyParView is used in Partisan, the high-performance, high-

scalability distributed computing framework. Plumtree was used in the Riak core for cluster-wide information.

Further Reading

If you'd like to learn more about the concepts mentioned in this chapter, you can refer to the following sources:

Gossip protocols

Shah, Devavrat. 2009. "Gossip Algorithms." *Foundations and Trends in Networking* 3, no. 1 (January): 1-125. *https://doi.org/10.1561/1300000014*.

Jelasity, Márk. 2003. "Gossip-based Protocols for Large-scale Distributed Systems." Dissertation. *http://www.inf.u-szeged.hu/~jelasity/dr/doktori-mu.pdf*.

Demers, Alan, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. 1987. "Epidemic algorithms for replicated database maintenance." In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing (PODC '87)*, 1-12. New York: Association for Computing Machinery. *https://doi.org/10.1145/41840.41841*.

CHAPTER 13
Distributed Transactions

To maintain order in a distributed system, we have to guarantee at least some consistency. In "Consistency Models" on page 222, we talked about single-object, single-operation consistency models that help us to reason about the individual operations. However, in databases we often need to execute *multiple* operations atomically.

Atomic operations are explained in terms of state transitions: the database was in state `A` before a particular transaction was started; by the time it finished, the state went from `A` to `B`. In operation terms, this is simple to understand, since transactions have no predetermined attached state. Instead, they apply operations to data records starting at *some* point in time. This gives us some flexibility in terms of scheduling and execution: transactions can be reordered and even retried.

The main focus of transaction processing is to determine permissible *histories*, to model and represent possible interleaving execution scenarios. History, in this case, represents a dependency graph: which transactions have been executed prior to execution of the current transaction. History is said to be *serializable* if it is equivalent (i.e., has the same dependency graph) to *some* history that executes these transactions sequentially. You can review concepts of histories, their equivalence, serializability, and other concepts in "Serializability" on page 94. Generally, this chapter is a distributed systems counterpart of Chapter 5, where we discussed node-local transaction processing.

Single-partition transactions involve the pessimistic (lock-based or tracking) or optimistic (try and validate) concurrency control schemes that we discussed in Chapter 5, but neither one of these approaches solves the problem of multipartition transactions, which require coordination between different servers, distributed commit, and rollback protocols.

Generally speaking, when transferring money from one account to another, you'd like to both credit the first account and debit the second one *simultaneously*. However, if we break down the transaction into individual steps, even debiting or crediting doesn't look atomic at first sight: we need to read the old balance, add or subtract the required amount, and save this result. Each one of these substeps involves several operations: the node receives a request, parses it, locates the data on disk, makes a write and, finally, acknowledges it. Even this is a rather high-level view: to execute a simple write, we have to perform hundreds of small steps.

This means that we have to first *execute* the transaction and only then make its results *visible*. But let's first define what transactions are. A *transaction* is a set of operations, an atomic unit of execution. Transaction atomicity implies that all its results become visible or none of them do. For example, if we modify several rows, or even tables in a single transaction, either all or none of the modifications will be applied.

To ensure atomicity, transactions should be *recoverable*. In other words, if the transaction cannot complete, is aborted, or times out, its results have to be rolled back completely. A nonrecoverable, partially executed transaction can leave the database in an inconsistent state. In summary, in case of unsuccessful transaction execution, the database state has to be reverted to its previous state, as if this transaction was never tried in the first place.

Another important aspect is network partitions and node failures: nodes in the system fail and recover independently, but their states have to remain consistent. This means that the atomicity requirement holds not only for the local operations, but also for operations executed on other nodes: changes have to be durably propagated to all of the nodes involved in the transaction or none of them [LAMPSON79].

Making Operations Appear Atomic

To make multiple operations appear atomic, especially if some of them are remote, we need to use a class of algorithms called *atomic commitment*. Atomic commitment doesn't allow disagreements between the participants: a transaction *will not* commit if even one of the participants votes against it. At the same time, this means that *failed* processes have to reach the same conclusion as the rest of the cohort. Another important implication of this fact is that atomic commitment algorithms do not work in the presence of Byzantine failures: when the process lies about its state or decides on an arbitrary value, since it contradicts unanimity [HADZILACOS05].

The problem that atomic commitment is trying to solve is reaching an agreement on whether or not to execute the proposed transaction. Cohorts cannot choose, influence, or change the proposed transaction or propose any alternative: they can only give their vote on whether or not they are willing to execute it [ROBINSON08].

Atomic commitment algorithms do not set strict requirements for the semantics of transaction *prepare*, *commit*, or *rollback* operations. Database implementers have to decide on:

- When the data is considered ready to commit, and they're just a pointer swap away from making the changes public.
- How to perform the commit itself to make transaction results visible in the shortest timeframe possible.
- How to roll back the changes made by the transaction if the algorithm decides not to commit.

We discussed node-local implementations of these processes in Chapter 5.

Many distributed systems use atomic commitment algorithms—for example, MySQL (for distributed transactions) and Kafka (for producer and consumer interaction [MEHTA17]).

In databases, distributed transactions are executed by the component commonly known as a *transaction manager*. The transaction manager is a subsystem responsible for scheduling, coordinating, executing, and tracking transactions. In a distributed environment, the transaction manager is responsible for ensuring that node-local visibility guarantees are consistent with the visibility prescribed by distributed atomic operations. In other words, transactions commit in all partitions, and for all replicas.

We will discuss two atomic commitment algorithms: two-phase commit, which solves a commitment problem, but doesn't allow for failures of the coordinator process; and three-phase commit [SKEEN83], which solves a *nonblocking atomic commitment* problem,[1] and allows participants proceed even in case of coordinator failures [BABAOGLU93].

Two-Phase Commit

Let's start with the most straightforward protocol for a distributed commit that allows multipartition *atomic* updates. (For more information on partitioning, you can refer to "Database Partitioning" on page 270.) *Two-phase commit* (2PC) is usually discussed in the context of database transactions. 2PC executes in two phases. During the first phase, the decided value is distributed, and votes are collected. During the second phase, nodes just flip the switch, making the results of the first phase visible.

1 The fine print says "assuming a highly reliable network." In other words, a network that precludes partitions [ALHOUMAILY10]. Implications of this assumption are discussed in the paper's section about algorithm description.

2PC assumes the presence of a *leader* (or *coordinator*) that holds the state, collects votes, and is a primary point of reference for the agreement round. The rest of the nodes are called *cohorts*. Cohorts, in this case, are usually partitions that operate over disjoint datasets, against which transactions are performed. The coordinator and every cohort keep local operation logs for each executed step. Participants vote to accept or reject some *value*, proposed by the coordinator. Most often, this value is an identifier of the distributed transaction that has to be executed, but 2PC can be used in other contexts as well.

The coordinator can be a node that received a request to execute the transaction, or it can be picked at random, using a leader-election algorithm, assigned manually, or even fixed throughout the lifetime of the system. The protocol does not place restrictions on the coordinator role, and the role can be transferred to another participant for reliability or performance.

As the name suggests, a two-phase commit is executed in two steps:

Prepare
: The coordinator notifies cohorts about the new transaction by sending a `Propose` message. Cohorts make a decision on whether or not they can commit the part of the transaction that applies to them. If a cohort decides that it can commit, it notifies the coordinator about the positive vote. Otherwise, it responds to the coordinator, asking it to abort the transaction. All decisions taken by cohorts are persisted in the coordinator log, and each cohort keeps a copy of its decision locally.

Commit/abort
: Operations within a transaction can change state across different partitions (each represented by a cohort). If even one of the cohorts votes to abort the transaction, the coordinator sends the `Abort` message to all of them. Only if all cohorts have voted positively does the coordinator send them a final `Commit` message.

This process is shown in Figure 13-1.

During the *prepare* phase, the coordinator distributes the proposed value and collects votes from the participants on whether or not this proposed value should be committed. Cohorts may choose to reject the coordinator's proposal if, for example, another conflicting transaction has already committed a different value.

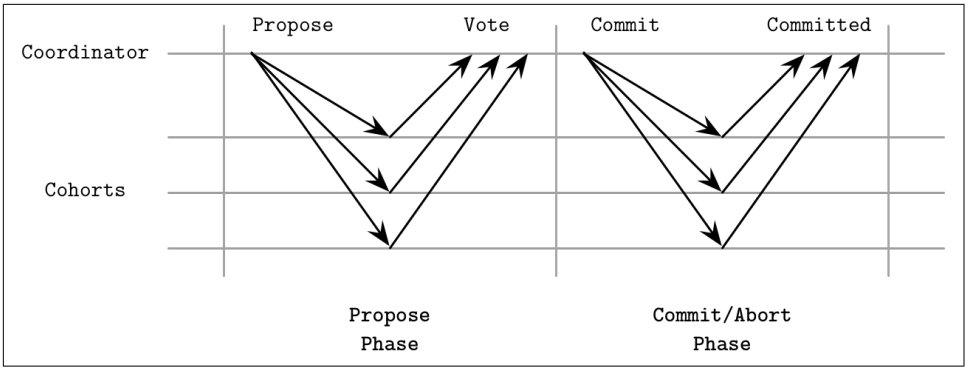

Figure 13-1. Two-phase commit protocol. During the first phase, cohorts are notified about the new transaction. During the second phase, the transaction is committed or aborted.

After the coordinator has collected the votes, it can make a decision on whether to *commit* the transaction or *abort* it. If all cohorts have voted positively, it decides to commit and notifies them by sending a `Commit` message. Otherwise, the coordinator sends an `Abort` message to all cohorts and the transaction gets rolled back. In other words, if one node rejects the proposal, the whole round is aborted.

During each step the coordinator and cohorts have to write the results of each operation to durable storage to be able to reconstruct the state and recover in case of local failures, and be able to forward and replay results for other participants.

In the context of database systems, each 2PC round is usually responsible for a single transaction. During the *prepare* phase, transaction contents (operations, identifiers, and other metadata) are transferred from the coordinator to the cohorts. The transaction is executed by the cohorts locally and is left in a *partially committed* state (sometimes called *precommitted*), making it ready for the coordinator to finalize execution during the next phase by either committing or aborting it. By the time the transaction commits, its contents are already stored durably on all other nodes [BERNSTEIN09].

Cohort Failures in 2PC

Let's consider several failure scenarios. For example, as Figure 13-2 shows, if one of the cohorts fails during the *propose* phase, the coordinator cannot proceed with a commit, since it requires all votes to be positive. If one of the cohorts is unavailable, the coordinator will abort the transaction. This requirement has a negative impact on availability: failure of a single node can prevent transactions from happening. Some systems, for example, Spanner (see "Distributed Transactions with Spanner" on page 268), perform 2PC over Paxos groups rather than individual nodes to improve protocol availability.

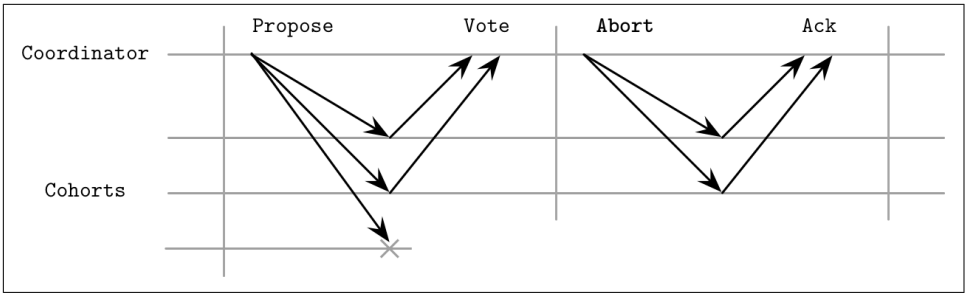

Figure 13-2. Cohort failure during the propose phase

The main idea behind 2PC is a *promise* by a cohort that, once it has positively responded to the proposal, it will not go back on its decision, so only the coordinator can abort the transaction.

If one of the cohorts has failed *after* accepting the proposal, it has to learn about the actual outcome of the vote before it can serve values correctly, since the coordinator might have aborted the commit due to the other cohorts' decisions. When a cohort node recovers, it has to get up to speed with a final coordinator decision. Usually, this is done by persisting the decision log on the coordinator side and replicating decision values to the failed participants. Until then, the cohort cannot serve requests because it is in an inconsistent state.

Since the protocol has multiple spots where processes are waiting for the other participants (when the coordinator collects votes, or when the cohort is waiting for the commit/abort phase), link failures might lead to message loss, and this wait will continue indefinitely. If the coordinator does not receive a response from the replica during the propose phase, it can trigger a timeout and abort the transaction.

Coordinator Failures in 2PC

If one of the cohorts does not receive a commit or abort command from the coordinator during the second phase, as shown in Figure 13-3, it should attempt to find out which decision was made by the coordinator. The coordinator might have decided upon the value but wasn't able to communicate it to the particular replica. In such cases, information about the decision can be replicated from the peers' transaction logs or from the backup coordinator. Replicating commit decisions is safe since it's always unanimous: the whole point of 2PC is to either commit or abort on all sites, and commit on one cohort implies that all other cohorts have to commit.

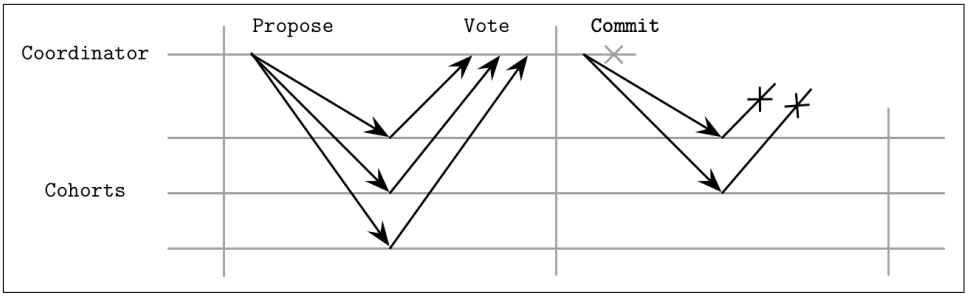

Figure 13-3. Coordinator failure after the propose phase

During the first phase, the coordinator collects votes and, subsequently, promises from cohorts, that they will wait for its explicit commit or abort command. If the coordinator fails after collecting the votes, but before broadcasting vote results, the cohorts end up in a state of uncertainty. This is shown in Figure 13-4. Cohorts do not know what precisely the coordinator has decided, and whether or not any of the participants (potentially also unreachable) might have been notified about the transaction results [BERNSTEIN87].

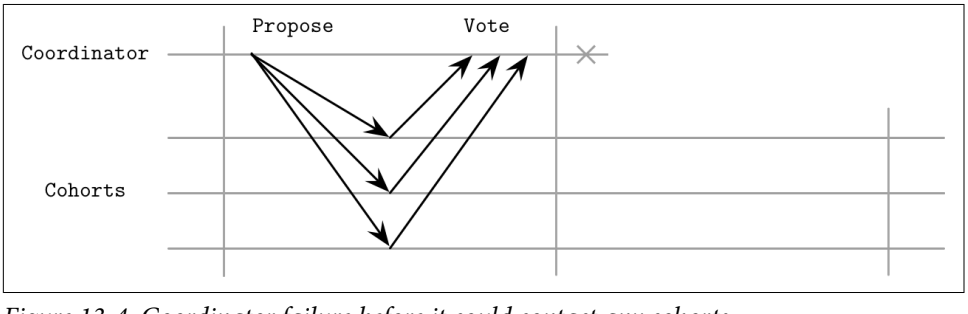

Figure 13-4. Coordinator failure before it could contact any cohorts

Inability of the coordinator to proceed with a commit or abort leaves the cluster in an undecided state. This means that cohorts will not be able to learn about the final decision in case of a permanent coordinator failure. Because of this property, we say that 2PC is a *blocking* atomic commitment algorithm. If the coordinator never recovers, its replacement has to collect votes for a given transaction again, and proceed with a final decision.

Many databases use 2PC: MySQL, PostgreSQL, MongoDB,[2] and others. Two-phase commit is often used to implement distributed transactions because of its simplicity (it is easy to reason about, implement, and debug) and low overhead (message com-

2 However, the documentation says that as of v3.6, 2PC provides only transaction-*like* semantics: *https://databass.dev/links/7*.

plexity and the number of round-trips of the protocol are low). It is important to implement proper recovery mechanisms and have backup coordinator nodes to reduce the chance of the failures just described.

Three-Phase Commit

To make an atomic commitment protocol robust against coordinator failures and avoid undecided states, the three-phase commit (3PC) protocol adds an extra step, and timeouts on *both* sides that can allow cohorts to proceed with either commit or abort in the event of coordinator failure, depending on the system state. 3PC assumes a synchronous model and that communication failures are not possible [BABAOGLU93].

3PC adds a *prepare* phase before the commit/abort step, which communicates cohort states collected by the coordinator during the propose phase, allowing the protocol to carry on even if the coordinator fails. All other properties of 3PC and a requirement to have a coordinator for the round are similar to its two-phase sibling. Another useful addition to 3PC is timeouts on the cohort side. Depending on which step the process is currently executing, either a commit or abort decision is forced on timeout.

As Figure 13-5 shows, the three-phase commit round consists of three steps:

Propose
: The coordinator sends out a proposed value and collects the votes.

Prepare
: The coordinator notifies cohorts about the vote results. If the vote has passed and all cohorts have decided to commit, the coordinator sends a `Prepare` message, instructing them to prepare to commit. Otherwise, an `Abort` message is sent and the round completes.

Commit
: Cohorts are notified by the coordinator to commit the transaction.

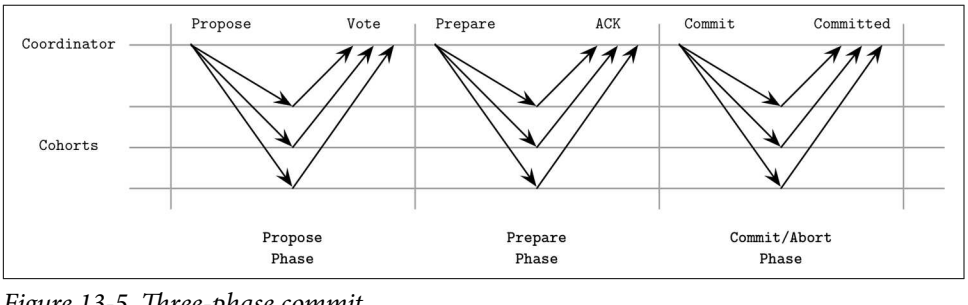

Figure 13-5. Three-phase commit

During the *propose* step, similar to 2PC, the coordinator distributes the proposed value and collects votes from cohorts, as shown in Figure 13-5. If the coordinator crashes during this phase and the operation times out, or if one of the cohorts votes negatively, the transaction will be aborted.

After collecting the votes, the coordinator makes a decision. If the coordinator decides to proceed with a transaction, it issues a `Prepare` command. It may happen that the coordinator cannot distribute prepare messages to all cohorts or it fails to receive their acknowledgments. In this case, cohorts may abort the transaction after timeout, since the algorithm hasn't moved all the way to the *prepared* state.

As soon as all the cohorts successfully move into the prepared state and the coordinator has received their prepare acknowledgments, the transaction will be committed if either side fails. This can be done since all participants at this stage have the same view of the state.

During *commit*, the coordinator communicates the results of the *prepare* phase to all the participants, resetting their timeout counters and effectively finishing the transaction.

Coordinator Failures in 3PC

All state transitions are coordinated, and cohorts can't move on to the next phase until everyone is done with the previous one: the coordinator has to wait for the replicas to continue. Cohorts can eventually abort the transaction if they do not hear from the coordinator before the timeout, if they didn't move past the prepare phase.

As we discussed previously, 2PC cannot recover from coordinator failures, and cohorts may get stuck in a nondeterministic state until the coordinator comes back. 3PC avoids blocking the processes in this case and allows cohorts to proceed with a deterministic decision.

The worst-case scenario for the 3PC is a network partition, shown in Figure 13-6. Some nodes successfully move to the prepared state, and now can proceed with commit after the timeout. Some can't communicate with the coordinator, and will abort after the timeout. This results in a split brain: some nodes proceed with a commit and some abort, all according to the protocol, leaving participants in an inconsistent and contradictory state.

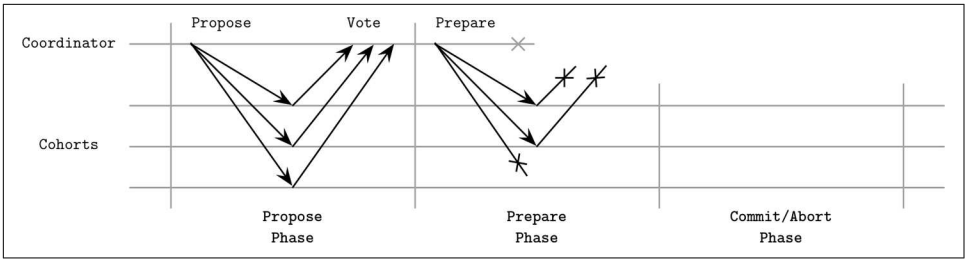

Figure 13-6. Coordinator failure during the second phase

While in theory 3PC does, to a degree, solve the problem with 2PC blocking, it has a larger message overhead, introduces potential contradictions, and does not work well in the presence of network partitions. This might be the primary reason 3PC is not widely used in practice.

Distributed Transactions with Calvin

We've already touched on the subject of synchronization costs and several ways around it. But there are other ways to reduce contention and the total amount of time during which transactions hold locks. One of the ways to do this is to let replicas agree on the execution order and transaction boundaries before acquiring locks and proceeding with execution. If we can achieve this, node failures do not cause transaction aborts, since nodes can recover state from other participants that execute the same transaction in parallel.

Traditional database systems execute transactions using two-phase locking or optimistic concurrency control and have no deterministic transaction order. This means that nodes have to be coordinated to preserve order. Deterministic transaction order removes coordination overhead during the execution phase and, since all replicas get the same inputs, they also produce equivalent outputs. This approach is commonly known as Calvin, a fast distributed transaction protocol [THOMSON12]. One of the prominent examples implementing distributed transactions using Calvin is FaunaDB.

To achieve deterministic order, Calvin uses a *sequencer*: an entry point for all transactions. The sequencer determines the order in which transactions are executed, and establishes a global transaction input sequence. To minimize contention and batch decisions, the timeline is split into *epochs*. The sequencer collects transactions and groups them into short time windows (the original paper mentions 10-millisecond batches), which also become replication units, so transactions do not have to be communicated separately.

As soon as a transaction batch is successfully replicated, sequencer forwards it to the *scheduler*, which orchestrates transaction execution. The scheduler uses a deterministic scheduling protocol that executes parts of transaction in parallel, while preserving the serial execution order specified by the sequencer. Since applying transaction to a specific state is guaranteed to produce only changes specified by the transaction and transaction order is predetermined, replicas do not have to further communicate with the sequencer.

Each transaction in Calvin has a *read set* (its dependencies, which is a collection of data records from the current database state required to execute it) and a *write set* (results of the transaction execution; in other words, its side effects). Calvin does not natively support transactions that rely on additional reads that would determine read and write sets.

A worker thread, managed by the scheduler, proceeds with execution in four steps:

1. It analyzes the transaction's read and write sets, determines node-local data records from the read set, and creates the list of *active* participants (i.e., ones that hold the elements of the write set, and will perform modifications on the data).
2. It collects the *local* data required to execute the transaction, in other words, the read set records that happen to reside on that node. The collected data records are forwarded to the corresponding *active* participants.
3. If this worker thread is executing on an active participant node, it receives data records forwarded from the other participants, as a counterpart of the operations executed during step 2.
4. Finally, it executes a batch of transactions, persisting results into local storage. It does not have to forward execution results to the other nodes, as they receive the same inputs for transactions and execute and persist results locally themselves.

A typical Calvin implementation colocates sequencer, scheduler, worker, and storage subsystems, as Figure 13-7 shows. To make sure that sequencers reach consensus on exactly which transactions make it into the current epoch/batch, Calvin uses the Paxos consensus algorithm (see "Paxos" on page 285) or asynchronous replication, in which a dedicated replica serves as a leader. While using a leader can improve latency, it comes with a higher cost of recovery as nodes have to reproduce the state of the failed leader in order to proceed.

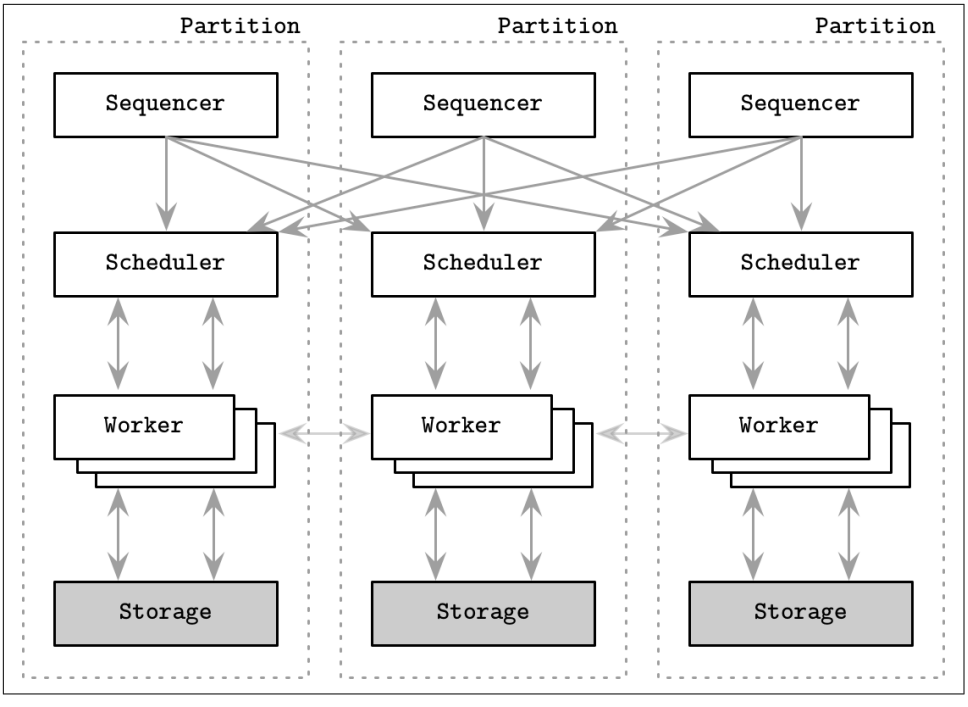

Figure 13-7. Calvin architecture

Distributed Transactions with Spanner

Calvin is often contrasted with another approach for distributed transaction management called Spanner [CORBETT12]. Its implementations (or derivatives) include several open source databases, most prominently CockroachDB and YugaByte DB. While Calvin establishes the global transaction execution order by reaching consensus on sequencers, Spanner uses two-phase commit over consensus groups per partition (in other words, per shard). Spanner has a rather complex setup, and we only cover high-level details in the scope of this book.

To achieve consistency and impose transaction order, Spanner uses *TrueTime*: a high-precision wall-clock API that also exposes an uncertainty bound, allowing local operations to introduce artificial slowdowns to wait for the uncertainty bound to pass.

Spanner offers three main operation types: *read-write transactions*, *read-only transactions*, and *snapshot reads*. Read-write transactions require locks, pessimistic concurrency control, and presence of the leader replica. Read-only transactions are lock-free and can be executed at any replica. A leader is required only for reads at the *latest* timestamp, which takes the latest committed value from the Paxos group. Reads at the specific timestamp are consistent, since values are versioned and snapshot contents can't be changed once written. Each data record has a timestamp assigned,

which holds a value of the transaction commit time. This also implies that multiple timestamped versions of the record can be stored.

Figure 13-8 shows the Spanner architecture. Each *spanserver* (replica, a server instance that serves data to clients) holds several *tablets*, with Paxos (see "Paxos" on page 285) state machines attached to them. Replicas are grouped into replica sets called Paxos groups, a unit of data placement and replication. Each Paxos group has a long-lived leader (see "Multi-Paxos" on page 291). Leaders communicate with each other during multishard transactions.

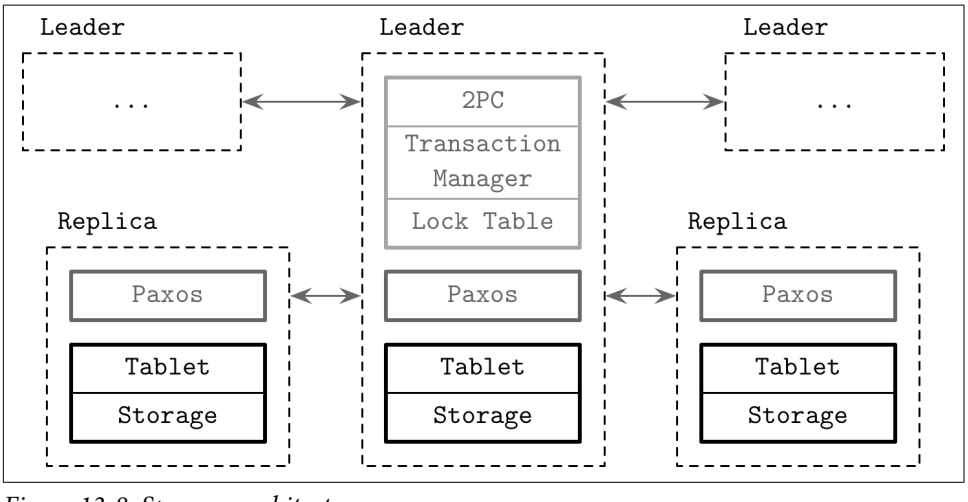

Figure 13-8. Spanner architecture

Every write has to go through the Paxos group leader, while reads can be served directly from the tablet on up-to-date replicas. The leader holds a *lock table* that is used to implement concurrency control using the two-phase locking (see "Lock-Based Concurrency Control" on page 100) mechanism and a *transaction manager* that is responsible for multishard distributed transactions. Operations that require synchronization (such as writes and reads within a transaction) have to acquire the locks from the lock table, while other operations (snapshot reads) can access the data directly.

For multishard transactions, group leaders have to coordinate and perform a two-phase commit to ensure consistency, and use two-phase locking to ensure isolation. Since the 2PC algorithm requires the presence of all participants for a successful commit, it hurts availability. Spanner solves this by using Paxos groups rather than individual nodes as cohorts. This means that 2PC can continue operating even if some of the members of the group are down. Within the Paxos group, 2PC contacts only the node that serves as a leader.

Paxos groups are used to consistently replicate transaction manager states across multiple nodes. The Paxos leader first acquires write locks, and chooses a write timestamp that is guaranteed to be larger than any previous transactions' timestamp, and records a 2PC `prepare` entry through Paxos. The transaction coordinator collects timestamps and generates a commit timestamp that is greater than any of the prepare timestamps, and logs a `commit` entry through Paxos. It then waits until *after* the timestamp it has chosen for commit, since it has to guarantee that clients will only see transaction results whose timestamps are in the past. After that, it sends this timestamp to the client and leaders, which log the `commit` record with the new timestamp in their local Paxos group and are now free to release the locks.

Single-shard transactions do not have to consult the transaction manager (and, subsequently, do not have to perform a cross-partition two-phase commit), since consulting a Paxos group and a lock table is enough to guarantee transaction order and consistency within the shard.

Spanner read-write transactions offer a serialization order called *external consistency*: transaction timestamps reflect serialization order, even in cases of distributed transactions. External consistency has real-time properties equivalent to linearizability: if transaction T_1 commits before T_2 starts, T_1's timestamp is smaller than the timestamp of T_2.

To summarize, Spanner uses Paxos for consistent transaction log replication, two-phase commit for cross-shard transactions, and TrueTime for deterministic transaction ordering. This means that multipartition transactions have a higher cost due to an additional two-phase commit round, compared to Calvin [ABADI17]. Both approaches are important to understand since they allow us to perform transactions in partitioned distributes data stores.

Database Partitioning

While discussing Spanner and Calvin, we've been using the term *partitioning* quite heavily. Let's now discuss it in more detail. Since storing all database records on a single node is rather unrealistic for the majority of modern applications, many databases use partitioning: a logical division of data into smaller manageable segments.

The most straightforward way to partition data is by splitting it into ranges and allowing *replica sets* to manage only specific ranges (partitions). When executing queries, clients (or query coordinators) have to route requests based on the *routing key* to the correct replica set for both reads and writes. This partitioning scheme is typically called *sharding*: every replica set acts as a single source for a subset of data.

To use partitions most effectively, they have to be sized, taking the load and value distribution into consideration. This means that frequently accessed, read/write heavy ranges can be split into smaller partitions to spread the load between them. At the

same time, if some value ranges are more dense than other ones, it might be a good idea to split them into smaller partitions as well. For example, if we pick *zip code* as a routing key, since the country population is unevenly spread, some zip code ranges can have more data (e.g., people and orders) assigned to them.

When nodes are added to or removed from the cluster, the database has to repartition the data to maintain the balance. To ensure consistent movements, we should relocate the data before we update the cluster metadata and start routing requests to the new targets. Some databases perform *auto-sharding* and relocate the data using placement algorithms that determine optimal partitioning. These algorithms use information about read, write loads, and amounts of data in each shard.

To find a target node from the routing key, some database systems compute a *hash* of the key, and use some form of mapping from the hash value to the node ID. One of the advantages of using the hash functions for determining replica placement is that it can help to reduce range hot-spotting, since hash values do not sort the same way as the original values. While two lexicographically close routing keys would be placed at the same replica set, using hashed values would place them on different ones.

The most straightforward way to map hash values to node IDs is by taking a remainder of the division of the hash value by the size of the cluster (modulo). If we have `N` nodes in the system, the target node ID is picked by computing `hash(v) modulo N`. The main problem with this approach is that whenever nodes are added or removed and the cluster size changes from `N` to `N'`, many values returned by `hash(v) modulo N'` will differ from the original ones. This means that most of the data will have to be moved.

Consistent Hashing

In order to mitigate this problem, some databases, such as Apache Cassandra and Riak (among others), use a different partitioning scheme called *consistent hashing*. As previously mentioned, routing key values are hashed. Values returned by the hash function are mapped to a *ring*, so that after the largest possible value, it wraps around to its smallest value. Each node gets its own position on the ring and becomes responsible for the *range* of values, between its predecessor's and its own positions.

Using consistent hashing helps to reduce the number of relocations required for maintaining balance: a change in the ring affects only the *immediate neighbors* of the leaving or joining node, and not an entire cluster. The word *consistent* in the definition implies that, when the hash table is resized, if we have `K` possible hash keys and `n` nodes, on average we have to relocate only `K/n` keys. In other words, a consistent hash function output changes minimally as the function range changes [KARGER97].

Distributed Transactions with Percolator

Coming back to the subject of distributed transactions, isolation levels might be difficult to reason about because of the allowed read and write anomalies. If serializability is not required by the application, one of the ways to avoid the write anomalies described in SQL-92 is to use a transactional model called *snapshot isolation* (SI).

Snapshot isolation guarantees that all reads made within the transaction are consistent with a snapshot of the database. The snapshot contains all values that were *committed before* the transaction's start timestamp. If there's a *write-write conflict* (i.e., when two concurrently running transactions attempt to make a write to the same cell), only one of them will commit. This characteristic is usually referred to as *first committer wins*.

Snapshot isolation prevents *read skew*, an anomaly permitted under the read-committed isolation level. For example, a sum of `x` and `y` is supposed to be `100`. Transaction `T1` performs an operation `read(x)`, and reads the value `70`. `T2` updates two values `write(x, 50)` and `write(y, 50)`, and commits. If `T1` attempts to run `read(y)`, and proceeds with transaction execution based on the value of `y` (`50`), newly committed by `T2`, it will lead to an inconsistency. The value of `x` that `T1` has read *before* `T2` committed and the new value of `y` aren't consistent with each other. Since snapshot isolation only makes values up to a specific timestamp visible for transactions, the new value of `y`, `50`, won't be visible to `T1` [BERENSON95].

Snapshot isolation has several convenient properties:

- It allows *only* repeatable reads of committed data.
- Values are consistent, as they're read from the snapshot at a specific timestamp.
- Conflicting writes are aborted and retried to prevent inconsistencies.

Despite that, histories under snapshot isolation are *not* serializable. Since only conflicting writes to the *same cells* are aborted, we can still end up with a *write skew* (see "Read and Write Anomalies" on page 95). Write skew occurs when two transactions modify disjoint sets of values, each preserving invariants for the data it writes. Both transactions are allowed to commit, but a combination of writes performed by these transactions may violate these invariants.

Snapshot isolation provides semantics that can be useful for many applications and has the major advantage of efficient reads, because no locks have to be acquired since snapshot data cannot be changed.

Percolator is a library that implements a transactional API on top of the distributed database *Bigtable* (see "Wide Column Stores" on page 15). This is a great example of building a transaction API on top of the existing system. Percolator stores data

records, committed data point locations (write metadata), and locks in different columns. To avoid race conditions and reliably lock tables in a single RPC call, it uses a conditional mutation Bigtable API that allows it to perform read-modify-write operations with a single remote call.

Each transaction has to consult the *timestamp oracle* (a source of clusterwide-consistent monotonically increasing timestamps) twice: for a transaction start timestamp, and during commit. Writes are buffered and committed using a client-driven two-phase commit (see "Two-Phase Commit" on page 259).

Figure 13-9 shows how the contents of the table change during execution of the transaction steps:

- a) Initial state. After the execution of the previous transaction, `TS1` is the latest timestamp for both accounts. No locks are held.
- b) The first phase, called *prewrite*. The transaction attempts to acquire locks for all cells written during the transaction. One of the locks is marked as *primary* and is used for client recovery. The transaction checks for the possible conflicts: if any other transaction has already written any data with a later timestamp or there are unreleased locks at any timestamp. If any conflict is detected, the transaction aborts.
- c) If all locks were successfully acquired and the possibility of conflict is ruled out, the transaction can continue. During the second phase, the client releases its locks, starting with the primary one. It publishes its write by replacing the lock with a write record, updating write metadata with the timestamp of the latest data point.

Since the client may fail while trying to commit the transaction, we need to make sure that partial transactions are finalized or rolled back. If a later transaction encounters an incomplete state, it should attempt to release the primary lock and commit the transaction. If the primary lock is already released, transaction contents *have to be* committed. Only one transaction can hold a lock at a time and all state transitions are atomic, so situations in which two transactions attempt to perform operations on the contents are not possible.

		Data	Locks	Write Metadata
Account1	TS2	-	-	TS1 is latest
	TS1	$100	-	-
Account2	TS2	-	-	TS1 is latest
	TS1	$200	-	-

a) Initial State before moving $150 from Account2 to Account1

		Data	Locks	Write Metadata
Account1	TS3	$250	Primary	-
	TS2	-	-	TS1 is latest
	TS1	$100	-	-
Account2	TS3	$50	Primary at Account1	-
	TS2	-	-	TS1 is latest
	TS1	$200	-	-

b) State after taking locks and updating accounts

		Data	Locks	Write Metadata
Account1	TS4	-	-	TS3 is latest
	TS3	$250	-	-
	TS2	-	-	TS1 is latest
	TS1	$100	-	-
Account2	TS4	-	-	TS3 is latest
	TS3	$50	Primary at Account1	-
	TS2	-	-	TS1 is latest
	TS1	$200	-	-

c) Transaction commit releases locks and updates metadata with latest timestamp

Figure 13-9. Percolator transaction execution steps. Transaction credits $150 from Account2 and debits it to Account1.

Snapshot isolation is an important and useful abstraction, commonly used in transaction processing. Since it simplifies semantics, precludes some of the anomalies, and opens up an opportunity to improve concurrency and performance, many MVCC systems offer this isolation level.

One of the examples of databases based on the Percolator model is TiDB ("Ti" stands for Titatium). TiDB is a strongly consistent, highly available, and horizontally scalable open source database, compatible with MySQL.

Coordination Avoidance

One more example, discussing costs of serializability and attempting to reduce the amount of coordination while still providing strong consistency guarantees, is coordination avoidance [BAILIS14b]. Coordination can be avoided, while preserving data integrity constraints, if operations are invariant confluent. Invariant Confluence (*I*-Confluence) is defined as a property that ensures that two invariant-valid but diverged database states can be merged into a single valid, final state. Invariants in this case preserve consistency in ACID terms.

Because any two valid states can be merged into a valid state, *I*-Confluent operations can be executed without additional coordination, which significantly improves performance characteristics and scalability potential.

To preserve this invariant, in addition to defining an operation that brings our database to the new state, we have to define a *merge* function that accepts two states. This function is used in case states were updated independently and bring diverged states back to convergence.

Transactions are executed against the local database versions (snapshots). If a transaction requires any state from other partitions for execution, this state is made available for it locally. If a transaction commits, resulting changes made to the local snapshot are migrated and merged with the snapshots on the other nodes. A system model that allows coordination avoidance has to guarantee the following properties:

Global validity
: Required invariants are always satisfied, for both merged and divergent committed database states, and transactions cannot observe invalid states.

Availability
: If all nodes holding states are reachable by the client, the transaction has to reach a commit decision, or abort, if committing it would violate one of the transaction invariants.

Convergence
: Nodes can maintain their local states independently, but in the absence of further transactions and indefinite network partitions, they have to be able to reach the same state.

Coordination freedom
: Local transaction execution is independent from the operations against the local states performed on behalf of the other nodes.

One of the examples of implementing coordination avoidance is Read-Atomic Multi Partition (RAMP) transactions [BAILIS14c]. RAMP uses multiversion concurrency control and metadata of current in-flight operations to fetch any missing state

updates from other nodes, allowing read and write operations to be executed concurrently. For example, readers that overlap with some writer modifying the same entry can be detected and, if necessary, *repaired* by retrieving required information from the in-flight write metadata in an additional round of communication.

Using lock-based approaches in a distributed environment might be not the best idea, and instead of doing that, RAMP provides two properties:

Synchronization independence
: One client's transactions won't stall, abort, or force the other client's transactions to wait.

Partition independence
: Clients do not have to contact partitions whose values aren't involved in their transactions.

RAMP introduces the *read atomic* isolation level: transactions cannot observe any in-process state changes from in-flight, uncommitted, and aborted transactions. In other words, all (or none) transaction updates are visible to concurrent transactions. By that definition, the read atomic isolation level also precludes *fractured reads*: when a transaction observes only a subset of writes executed by some other transaction.

RAMP offers atomic write visibility without requiring mutual exclusion, which other solutions, such as distributed locks, often couple together. This means that transactions can proceed without stalling each other.

RAMP distributes transaction metadata that allows reads to detect concurrent in-flight writes. By using this metadata, transactions can detect the presence of newer record versions, find and fetch the latest ones, and operate on them. To avoid coordination, all local commit decisions must also be valid globally. In RAMP, this is solved by requiring that, by the time a write becomes visible in one partition, writes from the same transaction in all other involved partitions are also visible for readers in those partitions.

To allow readers and writers to proceed without blocking other concurrent readers and writers, while maintaining the read atomic isolation level both locally and system-wide (in all other partitions modified by the committing transaction), writes in RAMP are installed and made visible using two-phase commit:

Prepare
: The first phase prepares and places writes to their respective target partitions without making them visible.

Commit/abort
: The second phase publishes the state changes made by the write operation of the committing transaction, making them available atomically across all partitions, or rolls back the changes.

RAMP allows multiple versions of the same record to be present at any given moment: latest value, in-flight uncommitted changes, and stale versions, overwritten by later transactions. Stale versions have to be kept around only for in-progress read requests. As soon as all concurrent readers complete, stale values can be discarded.

Making distributed transactions performant and scalable is difficult because of the coordination overhead associated with preventing, detecting, and avoiding conflicts for the concurrent operations. The larger the system, or the more transactions it attempts to serve, the more overhead it incurs. The approaches described in this section attempt to reduce the amount of coordination by using invariants to determine where coordination can be avoided, and only paying the full price if it's absolutely necessary.

Summary

In this chapter, we discussed several ways of implementing distributed transactions. First, we discussed two atomic commitment algorithms: two- and three-phase commits. The big advantage of these algorithms is that they're easy to understand and implement, but have several shortcomings. In 2PC, a coordinator (or at least its substitute) has to be alive for the length of the commitment process, which significantly reduces availability. 3PC lifts this requirement for some cases, but is prone to split brain in case of network partition.

Distributed transactions in modern database systems are often implemented using consensus algorithms, which we're going to discuss in the next chapter. For example, both Calvin and Spanner, discussed in this chapter, use Paxos.

Consensus algorithms are more involved than atomic commit ones, but have much better fault-tolerance properties, and decouple decisions from their initiators and allow participants to decide on *a value* rather than on whether or not to accept *the value* [GRAY04].

Further Reading

If you'd like to learn more about the concepts mentioned in this chapter, you can refer to the following sources:

Atomic commitment integration with local transaction processing and recovery subsystems

Silberschatz, Abraham, Henry F. Korth, and S. Sudarshan. 2010. *Database Systems Concepts* (6th Ed.). New York: McGraw-Hill.

Garcia-Molina, Hector, Jeffrey D. Ullman, and Jennifer Widom. 2008. *Database Systems: The Complete Book* (2nd Ed.). Boston: Pearson.

Recent progress in the area of distributed transactions (ordered chronologically; this list is not intended to be exhaustive)

Cowling, James and Barbara Liskov. 2012. "Granola: low-overhead distributed transaction coordination." In *Proceedings of the 2012 USENIX conference on Annual Technical Conference (USENIX ATC '12)*: 21-21. USENIX.

Balakrishnan, Mahesh, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D. Davis, Sriram Rao, Tao Zou, and Aviad Zuck. 2013. "Tango: distributed data structures over a shared log." In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*: 324-340.

Ding, Bailu, Lucja Kot, Alan Demers, and Johannes Gehrke. 2015. "Centiman: elastic, high performance optimistic concurrency control by watermarking." In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15)*: 262-275.

Dragojević, Aleksandar, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. "No compromises: distributed transactions with consistency, availability, and performance." In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*: 54-70.

Zhang, Irene, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. "Building consistent transactions with inconsistent replication." In *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP '15)*: 263-278.

CHAPTER 14
Consensus

We've discussed quite a few concepts in distributed systems, starting with basics, such as links and processes, problems with distributed computing; then going through failure models, failure detectors, and leader election; discussed consistency models; and we're finally ready to put it all together for a pinnacle of distributed systems research: distributed consensus.

Consensus algorithms in distributed systems allow multiple processes to reach an agreement on a value. FLP impossibility (see "FLP Impossibility" on page 189) shows that it is impossible to guarantee consensus in a completely asynchronous system in a bounded time. Even if message delivery is guaranteed, it is impossible for one process to know whether the other one has crashed or is running slowly.

In Chapter 9, we discussed that there's a trade-off between failure-detection accuracy and how quickly the failure can be detected. Consensus algorithms assume an asynchronous model and guarantee safety, while an external failure detector can provide information about other processes, guaranteeing liveness [CHANDRA96]. Since failure detection is not always fully accurate, there will be situations when a consensus algorithm waits for a process failure to be detected, or when the algorithm is restarted because some process is incorrectly suspected to be faulty.

Processes have to agree on some value proposed by one of the participants, even if some of them happen to crash. A process is said to be *correct* if hasn't crashed and continues executing algorithm steps. Consensus is extremely useful for putting events in a particular order, and ensuring consistency among the participants. Using consensus, we can have a system where processes move from one value to the next one without losing certainty about which values the clients observe.

From a theoretical perspective, consensus algorithms have three properties:

Agreement
: The decision value is the same for all *correct* processes.

Validity
: The decided value was proposed by one of the processes.

Termination
: All *correct* processes eventually reach the decision.

Each one of these properties is extremely important. The agreement is embedded in the human understanding of consensus. The dictionary definition of consensus has the word "unanimity" in it. This means that upon the agreement, no process is allowed to have a different opinion about the outcome. Think of it as an agreement to meet at a particular time and place with your friends: all of you would like to meet, and only the specifics of the event are being agreed upon.

Validity is essential, because without it consensus can be trivial. Consensus algorithms require all processes to agree on some value. If processes use some predetermined, arbitrary default value as a decision output regardless of the proposed values, they will reach unanimity, but the output of such an algorithm will not be valid and it wouldn't be useful in reality.

Without termination, our algorithm will continue forever without reaching any conclusion or will wait indefinitely for a crashed process to come back, which is not very useful, either. Processes have to agree eventually and, for a consensus algorithm to be practical, this has to happen rather quickly.

Broadcast

A *broadcast* is a communication abstraction often used in distributed systems. Broadcast algorithms are used to disseminate information among a set of processes. There exist many broadcast algorithms, making different assumptions and providing different guarantees. Broadcast is an important primitive and is used in many places, including consensus algorithms. We've discussed one of the forms of broadcast—gossip dissemination—already (see "Gossip Dissemination" on page 250).

Broadcasts are often used for database replication when a single coordinator node has to distribute the data to all other participants. However, making this process reliable is not a trivial matter: if the coordinator crashes after distributing the message to some nodes but not the other ones, it leaves the system in an inconsistent state: some of the nodes observe a new message and some do not.

The simplest and the most straightforward way to broadcast messages is through a *best effort broadcast* [CACHIN11]. In this case, the sender is responsible for ensuring

message delivery to all the targets. If it fails, the other participants do not try to rebroadcast the message, and in the case of coordinator crash, this type of broadcast will fail silently.

For a broadcast to be *reliable*, it needs to guarantee that all correct processes receive the same messages, even if the sender crashes during transmission.

To implement a naive version of a reliable broadcast, we can use a failure detector and a fallback mechanism. The most straightforward fallback mechanism is to allow every process that received the message to forward it to every other process it's aware of. When the source process fails, other processes detect the failure and continue broadcasting the message, effectively *flooding* the network with N^2 messages (as shown in Figure 14-1). Even if the sender has crashed, messages still are picked up and delivered by the rest of the system, improving its reliability, and allowing all receivers to see the same messages [CACHIN11].

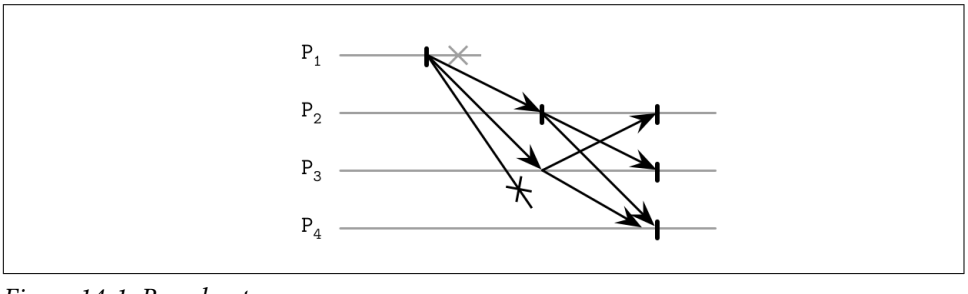

Figure 14-1. Broadcast

One of the downsides of this approach is the fact that it uses N^2 messages, where N is the number of *remaining* recipients (since every broadcasting process excludes the original process and itself). Ideally, we'd want to reduce the number of messages required for a reliable broadcast.

Atomic Broadcast

Even though the flooding algorithm just described can ensure message delivery, it does not guarantee delivery in any particular order. Messages reach their destination eventually, at an unknown time. If we need to deliver messages in order, we have to use the *atomic broadcast* (also called the *total order multicast*), which guarantees both reliable delivery and total order.

While a reliable broadcast ensures that the processes agree on the set of messages delivered, an atomic broadcast also ensures they agree on the same sequence of messages (i.e., message delivery order is the same for every target).

In summary, an atomic broadcast has to ensure two essential properties:

Atomicity
: Processes have to agree on the set of received messages. Either all nonfailed processes deliver the message, or none do.

Order
: All nonfailed processes deliver the messages in the same order.

Messages here are delivered *atomically*: every message is either delivered to all processes or none of them and, if the message is delivered, every other message is ordered before or after this message.

Virtual Synchrony

One of the frameworks for group communication using broadcast is called *virtual synchrony*. An atomic broadcast helps to deliver totally ordered messages to a *static* group of processes, and virtual synchrony delivers totally ordered messages to a *dynamic* group of peers.

Virtual synchrony organizes processes into groups. As long as the group exists, messages are delivered to all of its members in the same order. In this case, the order is not specified by the model, and some implementations can take this to their advantage for performance gains, as long as the order they provide is consistent across all members [BIRMAN10].

Processes have the same view of the group, and messages are associated with the group identity: processes can see the identical messages only as long as they belong to the same group.

As soon as one of the participants joins, leaves the group, or fails and is forced out of it, the group view changes. This happens by announcing the group change to all its members. Each message is uniquely associated with the group it has originated from.

Virtual synchrony distinguishes between the message *receipt* (when a group member receives the message) and its *delivery* (which happens when all the group members receive the message). If the message was *sent* in one view, it can be *delivered* only in the same view, which can be determined by comparing the current group with the group the message is associated with. Received messages remain pending in the queue until the process is notified about successful delivery.

Since every message belongs to a specific group, unless all processes in the group have *received* it before the view change, no group member can consider this message *delivered*. This implies that all messages are sent and delivered *between* the view changes, which gives us atomic delivery guarantees. In this case, group views serve as a barrier that message broadcasts cannot pass.

Some total broadcast algorithms order messages by using a single process (sequencer) that is responsible for determining it. Such algorithms can be easier to implement,

but rely on detecting the leader failures for liveness. Using a sequencer can improve performance, since we do not need to establish consensus between processes for every message, and can use a sequencer-local view instead. This approach can still scale by partitioning the requests.

Despite its technical soundness, virtual synchrony has not received broad adoption and isn't commonly used in end-user commercial systems [BIRMAN06].

Zookeeper Atomic Broadcast (ZAB)

One of the most popular and widely known implementations of the atomic broadcast is ZAB used by Apache Zookeeper [HUNT10] [JUNQUEIRA11], a hierarchical distributed key-value store, where it's used to ensure the total order of events and atomic delivery necessary to maintain consistency between the replica states.

Processes in ZAB can take on one of two roles: *leader* and *follower*. Leader is a temporary role. It drives the process by executing algorithm steps, broadcasts messages to followers, and establishes the event order. To write new records and execute reads that observe the most recent values, clients connect to one of the nodes in the cluster. If the node happens to be a leader, it will handle the request. Otherwise, it forwards the request to the leader.

To guarantee leader uniqueness, the protocol timeline is split into *epochs*, identified with a unique monotonically- and incrementally-sequenced number. During any epoch, there can be only one leader. The process starts from finding a *prospective leader* using any election algorithm, as long as it chooses a process that is up with a high probability. Since safety is guaranteed by the further algorithm steps, determining a prospective leader is more of a performance optimization. A prospective leader can also emerge as a consequence of the previous leader's failure.

As soon as a prospective leader is established, it executes a protocol in three phases:

Discovery
: The prospective leader learns about the latest epoch known by every other process, and proposes a new epoch that is *greater* than the current epoch of any follower. Followers respond to the epoch proposal with the identifier of the latest transaction seen in the previous epoch. After this step, no process will accept broadcast proposals for the earlier epochs.

Synchronization
: This phase is used to recover from the previous leader's failure and bring lagging followers up to speed. The prospective leader sends a message to the followers proposing itself as a leader for the new epoch and collects their acknowledgments. As soon as acknowledgments are received, the leader is established. After this step, followers will not accept attempts to become the epoch leader from any other processes. During synchronization, the new leader ensures that followers

have the same history and delivers committed proposals from the established leaders of earlier epochs. These proposals are delivered *before* any proposal from the new epoch is delivered.

Broadcast
: As soon as the followers are back in sync, active messaging starts. During this phase, the leader receives client messages, establishes their order, and broadcasts them to the followers: it sends a new proposal, waits for a quorum of followers to respond with acknowledgments and, finally, commits it. This process is similar to a two-phase commit without aborts: votes are just acknowledgments, and the client cannot vote against a valid leader's proposal. However, proposals from the leaders from incorrect epochs should *not* be acknowledged. The broadcast phase continues until the leader crashes, is partitioned from the followers, or is suspected to be crashed due to the message delay.

Figure 14-2 shows the three phases of the ZAB algorithm, and messages exchanged during each step.

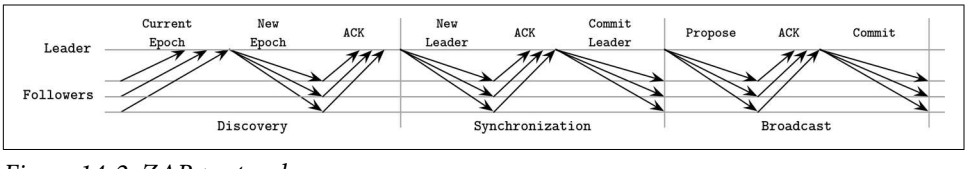

Figure 14-2. ZAB protocol summary

The safety of this protocol is guaranteed if followers ensure they accept proposals only from the leader of the established epoch. Two processes may *attempt* to get elected, but only one of them can win and establish itself as an epoch leader. It is also assumed that processes perform the prescribed steps in good faith and follow the protocol.

Both the leader and followers rely on heartbeats to determine the liveness of the remote processes. If the leader does not receive heartbeats from the quorum of followers, it steps down as a leader, and restarts the election process. Similarly, if one of the followers has determined the leader crashed, it starts a new election process.

Messages are totally ordered, and the leader will not attempt to send the next message until the message that preceded it was acknowledged. Even if some messages are received by a follower more than once, their repeated application do not produce additional side effects, as long as delivery order is followed. ZAB is able to handle multiple outstanding concurrent state changes from clients, since a unique leader will receive write requests, establish the event order, and broadcast the changes.

Total message order also allows ZAB to improve recovery efficiency. During the synchronization phase, followers respond with a highest committed proposal. The leader

can simply choose the node with the highest proposal for recovery, and this can be the only node messages have to be copied from.

One of the advantages of ZAB is its efficiency: the broadcast process requires only two rounds of messages, and leader failures can be recovered from by streaming the missing messages from a single up-to-date process. Having a long-lived leader can have a positive impact on performance: we do not require additional consensus rounds to establish a history of events, since the leader can sequence them based on its local view.

Paxos

An atomic broadcast is a problem equivalent to consensus in an asynchronous system with crash failures [CHANDRA96], since participants have to *agree* on the message order and must be able to learn about it. You will see many similarities in both motivation and implementation between atomic broadcast and consensus algorithms.

Probably the most widely known consensus algorithm is *Paxos*. It was first introduced by Leslie Lamport in "The Part-Time Parliament" paper [LAMPORT98]. In this paper, consensus is described in terms of terminology inspired by the legislative and voting process on the Aegian island of Paxos. In 2001, the author released a follow-up paper titled "Paxos Made Simple" [LAMPORT01] that introduced simpler terms, which are now commonly used to explain this algorithm.

Participants in Paxos can take one of three roles: *proposers*, *acceptors*, or *learners*:

Proposers
: Receive values from clients, create proposals to accept these values, and attempt to collect votes from acceptors.

Acceptors
: Vote to accept or reject the values proposed by the proposer. For fault tolerance, the algorithm requires the presence of multiple acceptors, but for liveness, only a quorum (majority) of acceptor votes is required to accept the proposal.

Learners
: Take the role of replicas, storing the outcomes of the accepted proposals.

Any participant can take any role, and most implementations colocate them: a single process can simultaneously be a proposer, an acceptor, and a learner.

Every proposal consists of a *value*, proposed by the client, and a unique monotonically increasing proposal number. This number is then used to ensure a total order of executed operations and establish happened-before/after relationships among them. Proposal numbers are often implemented using an `(id, timestamp)` pair, where node IDs are also comparable and can be used to break ties for timestamps.

Paxos Algorithm

The Paxos algorithm can be generally split into two phases: *voting* (or *propose* phase) and *replication*. During the voting phase, proposers compete to establish their leadership. During replication, the proposer distributes the value to the acceptors.

The proposer is an initial point of contact for the client. It receives a value that should be decided upon, and attempts to collect votes from the quorum of acceptors. When this is done, acceptors distribute the information about the agreed value to the learners, ratifying the result. Learners increase the replication factor of the value that's been agreed on.

Only one proposer can collect the majority of votes. Under some circumstances, votes may get split evenly between the proposers, and neither one of them will be able to collect a majority during this round, forcing them to restart. We discuss this and other scenarios of competing proposers in "Failure Scenarios" on page 288.

During the propose phase, the *proposer* sends a `Prepare(n)` message (where `n` is a proposal number) to a majority of acceptors and attempts to collect their votes.

When the *acceptor* receives the prepare request, it has to respond, preserving the following invariants [LAMPORT01]:

- If this acceptor hasn't responded to a prepare request with a higher sequence number yet, it *promises* that it will not accept any proposal with a lower sequence number.
- If this acceptor has already accepted (received an `Accept!(m,`$v_{accepted}$`)` message) any other proposal earlier, it responds with a `Promise(m,` $v_{accepted}$`)` message, notifying the proposer that it has already accepted the proposal with a sequence number `m`.
- If this acceptor has already responded to a prepare request with a higher sequence number, it notifies the proposer about the existence of a higher-numbered proposal.
- Acceptor can respond to more than one prepare request, as long as the later one has a higher sequence number .

During the replication phase, after collecting a majority of votes, the *proposer* can start the replication, where it commits the proposal by sending acceptors an `Accept!(n, v)` message with value `v` and proposal number `n`. `v` is the value associated with the highest-numbered proposal among the responses it received from acceptors, or any value of its own if their responses did not contain old accepted proposals.

The *acceptor* accepts the proposal with a number `n`, unless during the propose phase it has already responded to `Prepare(m)`, where `m` is greater than `n`. If the acceptor

rejects the proposal, it notifies the proposer about it by sending the highest sequence number it has seen along with the request to help the proposer catch up [LAMPORT01].

You can see a generalized depiction of a Paxos round in Figure 14-3.

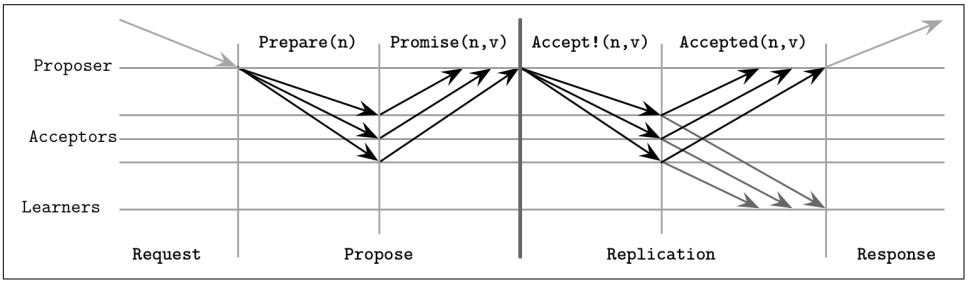

Figure 14-3. Paxos algorithm: normal execution

Once a consensus was reached on the value (in other words, it was accepted by at least one acceptor), future proposers have to decide on the same value to guarantee the agreement. This is why acceptors respond with the latest value they've accepted. If no acceptor has seen a previous value, the proposer is free to choose its own value.

A learner has to find out the value that has been decided, which it can know after receiving notification from the majority of acceptors. To let the learner know about the new value as soon as possible, acceptors can notify it about the value as soon as they accept it. If there's more than one learner, each acceptor will have to notify each learner. One or more learners can be *distinguished*, in which case it will notify other learners about accepted values.

In summary, the goal of the first algorithm phase is to establish a leader for the round and understand which value is going to be accepted, allowing the leader to proceed with the second phase: broadcasting the value. For the purpose of the base algorithm, we assume that we have to perform both phases every time we'd like to decide on a value. In practice, we'd like to reduce the number of steps in the algorithm, so we allow the proposer to propose more than one value. We discuss this in more detail later in "Multi-Paxos" on page 291.

Quorums in Paxos

Quorums are used to make sure that *some* of the participants can fail, but we still can proceed as long as we can collect votes from the alive ones. A *quorum* is the *minimum* number of votes required for the operation to be performed. This number usually constitutes a *majority* of participants. The main idea behind quorums is that even if participants fail or happen to be separated by the network partition, there's at least one participant that acts as an arbiter, ensuring protocol correctness.

Once a sufficient number of participants accept the proposal, the value is guaranteed to be accepted by the protocol, since any two majorities have at least one participant in common.

Paxos guarantees safety in the presence of any number of failures. There's no configuration that can produce incorrect or inconsistent states since this would contradict the definition of consensus.

Liveness is guaranteed in the presence of `f` failed processes. For that, the protocol requires `2f + 1` processes in total so that, if `f` processes happen to fail, there are still `f + 1` processes able to proceed. By using quorums, rather than requiring the presence of all processes, Paxos (and other consensus algorithms) guarantee results even when `f` process failures occur. In "Flexible Paxos" on page 296, we talk about quorums in slightly different terms and describe how to build protocols requiring quorum intersection between algorithm *steps* only.

It is important to remember that quorums only describe the blocking properties of the system. To guarantee safety, for each step we have to wait for responses from *at least* a quorum of nodes. We can send proposals and accept commands to more nodes; we just do not have to wait for their responses to proceed. We may send messages to more nodes (some systems use *speculative execution*: issuing redundant queries that help to achieve the required response count in case of node failures), but to guarantee liveness, we can proceed as soon as we hear from the quorum.

Failure Scenarios

Discussing distributed algorithms gets particularly interesting when failures are discussed. One of the failure scenarios, demonstrating fault tolerance, is when the proposer fails during the second phase, before it is able to broadcast the value to all the acceptors (a similar situation can happen if the proposer is alive but is slow or cannot communicate with some acceptors). In this case, the new proposer may pick up and commit the value, distributing it to the other participants.

Figure 14-4 shows this situation:

- Proposer P_1 goes through the election phase with a proposal number 1, but fails after sending the value V1 to just one acceptor A_1.
- Another proposer P_2 starts a new round with a higher proposal number 2, collects a quorum of acceptor responses (A_1 and A_2 in this case), and proceeds by committing the *old* value V1, proposed by P_1.

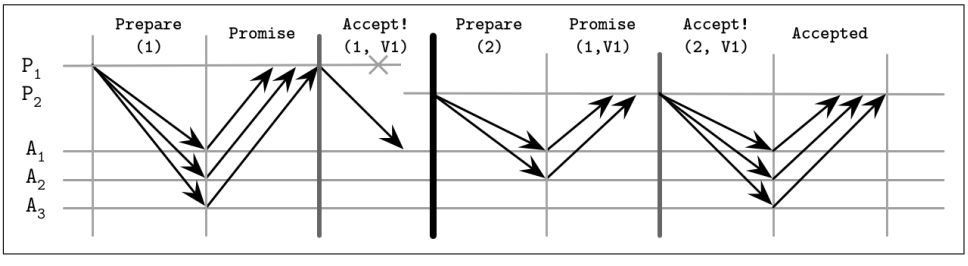

Figure 14-4. Paxos failure scenario: proposer failure, deciding on the old value

Since the algorithm state is replicated to multiple nodes, proposer failure does not result in failure to reach a consensus. If the current proposer fails after even a single acceptor A_1 has accepted the value, its proposal *can* be picked by the next proposer. This also implies that all of it may happen without the original proposer knowing about it.

In a client/server application, where the client is connected only to the original proposer, this might lead to situations where the client doesn't know about the result of the Paxos round execution.[1]

However, other scenarios are possible, too, as Figure 14-5 shows. For example:

- P_1 has failed just like in the previous example, after sending the value V1 only to A_1.
- The next proposer, P_2, starts a new round with a higher proposal number 2, and collects a quorum of acceptor responses, but this time A_2 and A_3 are first to respond. After collecting a quorum, P_2 commits *its own value* despite the fact that theoretically there's a different committed value on A_1.

1 For example, such a situation was described in *https://databass.dev/links/68*.

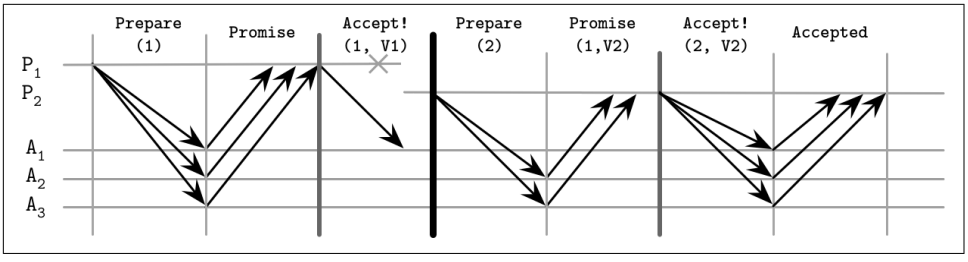

Figure 14-5. Paxos failure scenario: proposer failure, deciding on the new value

There's one more possibility here, shown in Figure 14-6:

- Proposer P_1 fails after only one acceptor A_1 accepts the value `V1`. A_1 fails shortly after accepting the proposal, before it can notify the next proposer about its value.
- Proposer P_2, which started the round after P_1 failed, does not overlap with A_1 and proceeds to commit its value instead.
- Any proposer that comes *after* this round that will overlap with A_1, will ignore A_1's value and choose a more recent accepted proposal instead.

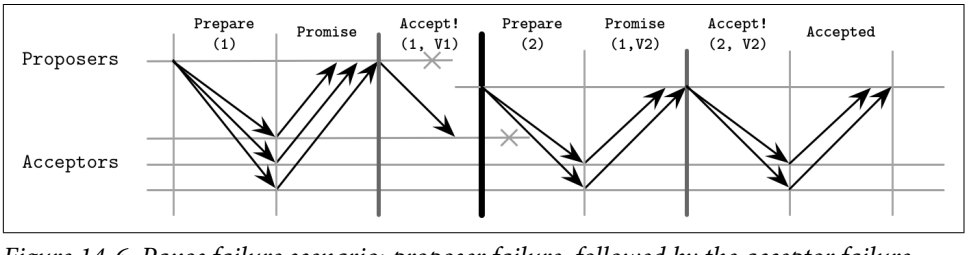

Figure 14-6. Paxos failure scenario: proposer failure, followed by the acceptor failure

Another failure scenario is when two or more proposers start competing, each trying to get through the propose phase, but keep failing to collect a majority because the other one beat them to it.

While acceptors promise not to accept any proposals with a lower number, they still may respond to multiple prepare requests, as long as the later one has a higher sequence number. When a proposer tries to commit the value, it might find that acceptors have already responded to a prepare request with a higher sequence number. This may lead to multiple proposers constantly retrying and preventing each other from further progress. This problem is usually solved by incorporating a random backoff, which eventually lets one of the proposers proceed while the other one sleeps.

The Paxos algorithm can tolerate acceptor failures, but only if there are still enough acceptors alive to form a majority.

Multi-Paxos

So far we discussed the classic Paxos algorithm, where we pick an arbitrary proposer and attempt to start a Paxos round. One of the problems with this approach is that a propose round is required for each replication round that occurs in the system. Only after the proposer is established for the round, which happens after a majority of acceptors respond with a `Promise` to the proposer's `Prepare`, can it start the replication. To avoid repeating the propose phase and let the proposer reuse its recognized position, we can use Multi-Paxos, which introduces the concept of a *leader*: a *distinguished proposer* [LAMPORT01]. This is a crucial addition, significantly improving algorithm efficiency.

Having an established leader, we can skip the propose phase and proceed straight to replication: distributing a value and collecting acceptor acknowledgments.

In the classic Paxos algorithm, reads can be implemented by running a Paxos round that would collect any values from incomplete rounds if they're present. This has to be done because the last known proposer is not guaranteed to hold the most recent data, since there might have been a different proposer that has modified state without the proposer knowing about it.

A similar situation may occur in Multi-Paxos: we're trying to perform a read from the known leader *after* the other leader is already elected, returning stale data, which contradicts the linearizability guarantees of consensus. To avoid that and guarantee that no other process can successfully submit values, some Multi-Paxos implementations use *leases*. The leader periodically contacts the participants, notifying them that it is still alive, effectively prolonging its lease. Participants have to respond and allow the leader to continue operation, promising that they will not accept proposals from other leaders for the period of the lease [CHANDRA07].

Leases are not a correctness guarantee, but a performance optimization that allows reads from the active leader without collecting a quorum. To guarantee safety, leases rely on the bounded clock synchrony between the participants. If their clocks drift too much and the leader assumes its lease is still valid while other participants think its lease has expired, linearizability *cannot* be guaranteed.

Multi-Paxos is sometimes described as a *replicated log* of operations applied to some structure. The algorithm is oblivious to the semantics of this structure and is only concerned with consistently replicating values that will be appended to this log. To preserve the state in case of process crashes, participants keep a durable log of received messages.

To prevent a log from growing indefinitely large, its contents should be applied to the aforementioned structure. After the log contents are synchronized with a primary structure, creating a snapshot, the log can be truncated. Log and state snapshots should be mutually consistent, and snapshot changes should be applied atomically with truncation of the log segment [CHANDRA07].

We can think of single-decree Paxos as a *write-once register*: we have a slot where we can put a value, and as soon as we've written the value there, no subsequent modifications are possible. During the first step, proposers compete for ownership of the register, and during the second phase, one of them writes the value. At the same time, Multi-Paxos can be thought of as an append-only log, consisting of a sequence of such values: we can write one value at a time, all values are strictly ordered, and we cannot modify already written values [RYSTSOV16]. There are examples of consensus algorithms that offer collections of read-modify-write registers and use state sharing rather than replicated state machines, such as Active Disk Paxos [CHOCKLER15] and CASPaxos [RYSTSOV18].

Fast Paxos

We can reduce the number of round-trips by one, compared to the classic Paxos algorithm, by letting *any* proposer contact acceptors directly rather than going through the leader. For this, we need to increase the quorum size to `2f + 1` (where `f` is the number of processes allowed to fail), compared to `f + 1` in classic Paxos, and a total number of acceptors to `3f + 1` [JUNQUEIRA07]. This optimization is called *Fast Paxos* [LAMPORT06].

The classic Paxos algorithm has a condition, where during the replication phase, the proposer can pick any value it has collected during the propose phase. Fast Paxos has two types of rounds: *classic*, where the algorithm proceeds the same way as the classic version, and *fast*, where it allows acceptors to accept other values.

While describing this algorithm, we will refer to the proposer that has collected a sufficient number of responses during the propose phase as a *coordinator*, and reserve term *proposer* for all other proposers. Some Fast Paxos descriptions say that *clients* can contact acceptors directly [ZHAO15].

In a fast round, if the coordinator is permitted to pick its own value during the replication phase, it can instead issue a special `Any` message to acceptors. Acceptors, in this case, are allowed to treat *any* proposer's value as if it is a classic round and they received a message with this value from the coordinator. In other words, acceptors independently decide on values they receive from different proposers.

Figure 14-7 shows an example of classic and fast rounds in Fast Paxos. From the image it might look like the fast round has more execution steps, but keep in mind that in a classic round, in order to submit its value, the proposer would need to go through the coordinator to get its value committed.

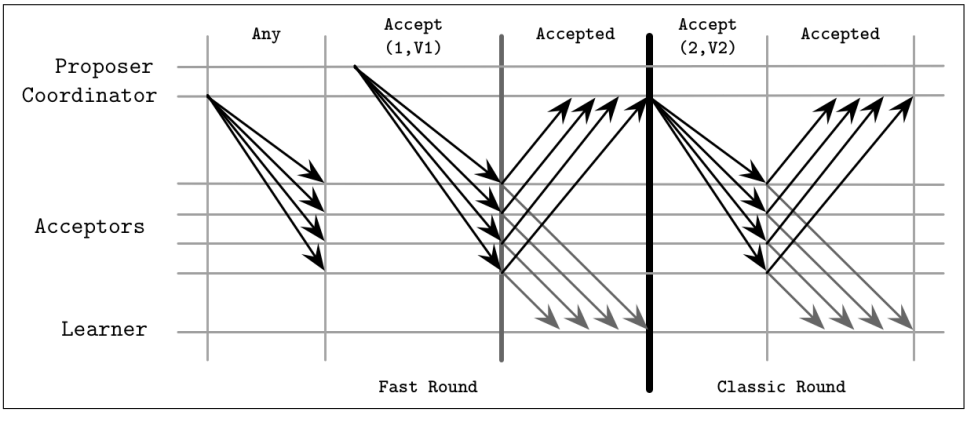

Figure 14-7. Fast Paxos algorithm: fast and classic rounds

This algorithm is prone to *collisions*, which occur if two or more proposers attempt to use the *fast* step and reduce the number of round-trips, and acceptors receive different values. The coordinator has to intervene and start recovery by initiating a new round.

This means that acceptors, after receiving values from different proposers, may decide on conflicting values. When the coordinator detects a conflict (value collision), it has to reinitiate a `Propose` phase to let acceptors converge to a single value.

One of the disadvantages of Fast Paxos is the increased number of round-trips and request latency on collisions if the request rate is high. [JUNQUEIRA07] shows that, due to the increased number of replicas and, subsequently, *messages* exchanged between the participants, despite a reduced number of steps, Fast Paxos can have higher latencies than its classic counterpart.

Egalitarian Paxos

Using a distinguished proposer as a leader makes a system prone to failures: as soon as the leader fails, the system has to elect a new one before it can proceed with further steps. Another problem is that having a leader can put a disproportionate load on it, impairing system performance.

One of the ways to avoid putting an entire system load on the leader is *partitioning*. Many systems split the range of possible values into smaller segments and allow a part of the system to be responsible for a specific range without having to worry about the other parts. This helps with availability (by isolating failures to a single partition and preventing propagation to other parts of the system), performance (since segments serving different values are nonoverlapping), and scalability (since we can scale the system by increasing the number of partitions). It is important to keep in mind that performing an operation against *multiple* partitions will require an atomic commitment.

Instead of using a leader and proposal numbers for sequencing commands, we can use a leader responsible for the commit of the *specific* command, and establish the order by looking up and setting dependencies. This approach is commonly called Egalitarian Paxos, or EPaxos [MORARU11]. The idea of allowing nonconflicting writes to be committed to the replicated state machine independently was first introduced in [LAMPORT05] and called Generalized Paxos. EPaxos is a first implementation of Generalized Paxos.

EPaxos attempts to offer benefits of both the classic Paxos algorithm and Multi-Paxos. Classic Paxos offers high availability, since a leader is established during each round, but has a higher message complexity. Multi-Paxos offers high throughput and requires fewer messages, but a leader may become a bottleneck.

EPaxos starts with a *Pre-Accept* phase, during which a process becomes a leader for the specific proposal. Every proposal has to include:

Dependencies
: All commands that potentially interfere with a current proposal, but are not necessarily already committed.

A sequence number
: This breaks cycles between the dependencies. Set it with a value larger than any sequence number of the known dependencies.

After collecting this information, it forwards a `Pre-Accept` message to a *fast quorum* of replicas. A fast quorum is $\lceil 3f/4 \rceil$ replicas, where `f` is the number of tolerated failures.

Replicas check their local command logs, update the proposal dependencies based on their view of potentially conflicting proposals, and send this information back to the leader. If the leader receives responses from a fast quorum of replicas, and their dependency lists are in agreement with each other and the leader itself, it can commit the command.

If the leader does not receive enough responses or if the command lists received from the replicas differ and contain interfering commands, it updates its proposal with a new dependency list and a sequence number. The new dependency list is based on previous replica responses and combines *all* collected dependencies. The new sequence number has to be larger than the highest sequence number seen by the replicas. After that, the leader sends the new, updated command to $\lfloor f/2 \rfloor + 1$ replicas. After this is done, the leader can finally commit the proposal.

Effectively, we have two possible scenarios:

Fast path
: When dependencies match and the leader can safely proceed with the commit phase with only a *fast quorum* of replicas.

Slow path
: When there's a disagreement between the replicas, and their command lists have to be updated before the leader can proceed with a commit.

Figure 14-8 shows these scenarios—P_1 initiating a fast path run, and P_5 initiating a slow path run:

- P_1 starts with proposal number `1` and no dependencies, and sends a `PreAccept(1, ∅)` message. Since the command logs of P_2 and P_3 are empty, P_1 can proceed with a commit.
- P_5 creates a proposal with sequence number `2`. Since its command log is empty by that point, it also declares no dependencies and sends a `PreAccept(2, ∅)` message. P_4 is not aware of the committed proposal `1`, but P_3 notifies P_5 about the conflict and sends its command log: `{1}`.
- P_5 updates its local dependency list and sends a message to make sure replicas have the same dependencies: `Accept(2,{1})`. As soon as the replicas respond, it can commit the value.

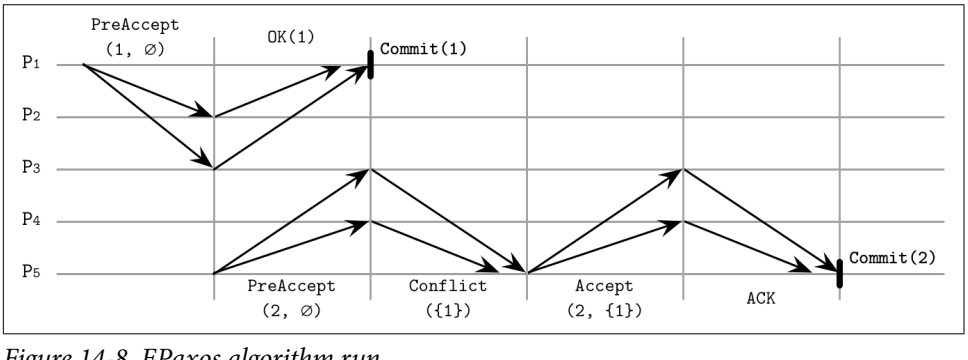

Figure 14-8. EPaxos algorithm run

Two commands, `A` and `B`, interfere only if their execution order matters; in other words, if executing `A` before `B` and executing `B` before `A` produce different results.

Commit is done by responding to the client and asynchronously notifying replicas with a `Commit` message. Commands are executed *after* they're committed.

Since dependencies are collected during the Pre-Accept phase, by the time requests are executed, the command order is already established and no command can suddenly appear somewhere in-between: it can only get appended *after* the command with the largest sequence number.

To execute a command, replicas build a dependency graph and execute all commands in a reverse dependency order. In other words, before a command can be executed, all its dependencies (and, subsequently, all their dependencies) have to be executed. Since only interfering commands have to depend on each other, this situation should be relatively rare for most workloads [MORARU13].

Similar to Paxos, EPaxos uses proposal numbers, which prevent stale messages from being propagated. Sequence numbers consist of an *epoch* (identifier of the current cluster configuration that changes when nodes leave and join the cluster), a monotonically incremented node-local counter, and a replica ID. If a replica receives a proposal with a sequence number lower than one it has already seen, it negatively acknowledges the proposal, and sends the highest sequence number and an updated command list known to it in response.

Flexible Paxos

A quorum is usually defined as a majority of processes. By definition, we have an *intersection* between two quorums no matter how we pick nodes: there's always at least one node that can break ties.

We have to answer two important questions:

- Is it necessary to contact the *majority* of servers during *every* execution step?
- Do *all* quorums have to intersect? In other words, does a quorum we use to pick a distinguished proposer (first phase), a quorum we use to decide on a value (second phase), and every execution instance (for example, if multiple instances of the second step are executed concurrently), have to have nodes in common?

Since we're still talking about consensus, we cannot change any safety definitions: the algorithm has to guarantee the agreement.

In Multi-Paxos, the leader election phase is infrequent, and the distinguished proposer is allowed to commit several values without rerunning the election phase, potentially staying in the lead for a longer period. In "Tunable Consistency" on page 235, we discussed formulae that help us to find configurations where we have

intersections between the node sets. One of the examples was to wait for just one node to acknowledge the write (and let the requests to the rest of nodes finish asynchronously), and read from *all* the nodes. In other words, as long as we keep `R + W > N`, there's at least one node in common between read and write sets.

Can we use a similar logic for consensus? It turns out that we can, and in Paxos we only require the group of nodes from the first phase (that elects a leader) to overlap with the group from the second phase (that participates in accepting proposals).

In other words, a quorum doesn't have to be defined as a majority, but only as a nonempty group of nodes. If we define a total number of participants as `N`, the number of nodes required for a propose phase to succeed as Q_1, and the number of nodes required for the accept phase to succeed as Q_2, we only need to ensure that $Q_1 + Q_2 > N$. Since the second phase is usually more common than the first one, Q_2 can contain only `N/2` acceptors, as long as Q_1 is adjusted to be correspondingly larger ($Q_1 = N - Q_2 + 1$). This finding is an important observation crucial for understanding consensus. The algorithm that uses this approach is called *Flexible Paxos* [HOWARD16].

For example, if we have five acceptors, as long as we require collecting votes from four of them to win the election round, we can allow the leader to wait for responses from two nodes during the replication stage. Moreover, since there's an overlap between *any* subset consisting of two acceptors with the leader election quorum, we can submit proposals to disjoint sets of acceptors. Intuitively, this works because whenever a new leader is elected without the current one being aware of it, there will always be at least one acceptor that knows about the existence of the new leader.

Flexible Paxos allows trading availability for latency: we reduce the number of nodes participating in the second phase but have to collect more votes, requiring more participants to be available during the leader election phase. The good news is that this configuration can continue the replication phase and tolerate failures of up to $N - Q_2$ nodes, as long as the current leader is stable and a new election round is not required.

Another Paxos variant using the idea of intersecting quorums is Vertical Paxos. Vertical Paxos distinguishes between read and write quorums. These quorums must intersect. A leader has to collect a smaller read quorum for one or more lower-numbered proposals, and a larger write quorum for its own proposal [LAMPORT09]. [LAMPSON01] also distinguishes between the *out* and *decision* quorums, which translate to prepare and accept phases, and gives a quorum definition similar to Flexible Paxos.

Generalized Solution to Consensus

Paxos might sometimes be a bit difficult to reason about: multiple roles, steps, and all the possible variations are hard to keep track of. But we can think of it in simpler terms. Instead of splitting roles between the participants and having decision rounds, we can use a simple set of concepts and rules to achieve guarantees of a single-decree

Paxos. We discuss this approach only briefly as this is a relatively new development [HOWARD19]—it's important to know, but we've yet to see its implementations and practical applications.

We have a client and a set of servers. Each server has multiple *registers*. A register has an index identifying it, can be written only once, and it can be in one of three states: unwritten, containing a *value*, and containing *nil* (a special empty value).

Registers with the same index located on different servers form a *register set*. Each register set can have one or more quorums. Depending on the state of the registers in it, a quorum can be in one of the *undecided* (`Any` and `Maybe v`), or *decided* (`None` and `Decided v`) states:

`Any`
: Depending on future operations, this quorum set can decide on any value.

`Maybe v`
: If this quorum reaches a decision, its decision can only be `v`.

`None`
: This quorum cannot decide on the value.

`Decided v`
: This quorum has decided on the value `v`.

The client exchanges messages with the servers and maintains a state table, where it keeps track of values and registers, and can infer decisions made by the quorums.

To maintain correctness, we have to limit how clients can interact with servers and which values they may write and which they may not. In terms of reading values, the client can output the decided value only if it has read it from the quorum of servers in the same register set.

The writing rules are slightly more involved because to guarantee algorithm safety, we have to preserve several invariants. First, we have to make sure that the client doesn't just come up with new values: it is allowed to write a specific value to the register only if it has received it as input or has read it from a register. Clients cannot write values that allow different quorums in the same register to decide on different values. Lastly, clients cannot write values that override previous decisions made in the previous register sets (decisions made in register sets up to `r - 1` have to be `None`, `Maybe v`, or `Decided v`).

Generalized Paxos algorithm

Putting all these rules together, we can implement a generalized Paxos algorithm that achieves consensus over a single value using write-once registers [HOWARD19]. Let's say we have three servers `[S0, S1, S2]`, registers `[R0, R1, …]`, and clients

`[C0, C1, ...]`, where the client can only write to the assigned subset of registers. We use simple majority quorums for all registers (`{S0, S1}`, `{S0, S2}`, `{S1, S2}`).

The decision process here consists of two phases. The first phase ensures that it is safe to write a value to the register, and the second phase writes the value to the register:

During phase 1
: The client checks if the register it is about to write is unwritten by sending a $P1_A$`(register)` command to the server. If the register is unwritten, all registers up to `register - 1` are set to `nil`, which prevents clients from writing to previous registers. The server responds with a set of registers written so far. If it receives responses from the majority of servers, the client chooses either the non-empty value from the register with the largest index or its own value in case no value is present. Otherwise, it restarts the first phase.

During phase 2
: The client notifies all servers about the value it has picked during the first phase by sending them $P2_A$`(register, value)`. If the majority of servers respond to this message, it can output the decision value. Otherwise, it starts again from phase 1.

Figure 14-9 shows this generalization of Paxos (adapted from [HOWARD19]). Client `C0` tries to commit value `V`. During the first step, its state table is empty, and servers `S0` and `S1` respond with the empty register set, indicating that no registers were written so far. During the second step, it can submit its value `V`, since no other value was written.

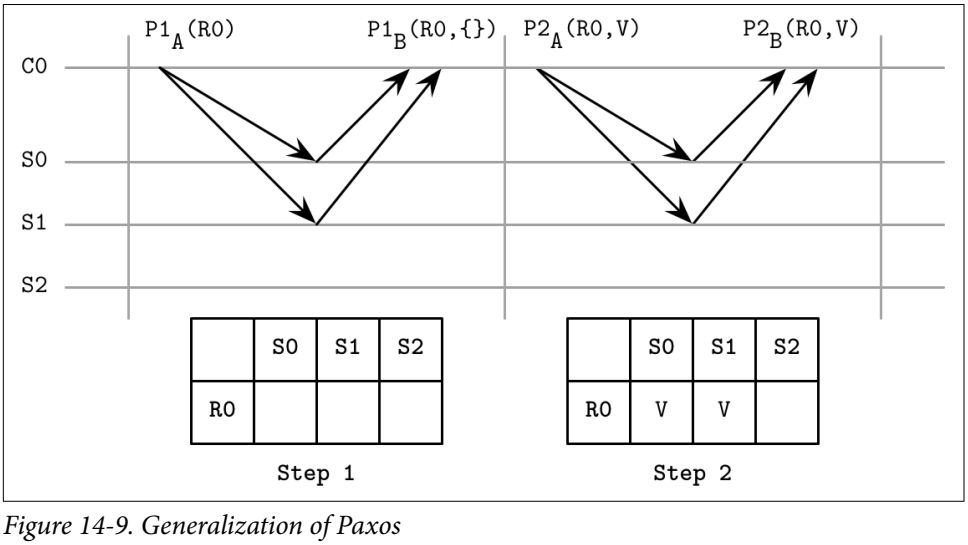

Figure 14-9. Generalization of Paxos

At that point, any other client can query servers to find out the current state. Quorum `{S0, S1}` has reached `Decided A` state, and quorums `{S0, S2}` and `{S1, S2}` have reached the `Maybe V` state for `R0`, so `C1` chooses the value `V`. At that point, no client can decide on a value other than `V`.

This approach helps to understand the semantics of Paxos. Instead of thinking about the state from the perspective of interactions of remote actors (e.g., a proposer finding out whether or not an acceptor has already accepted a different proposal), we can think in terms of the last known state, making our decision process simple and removing possible ambiguities. Immutable state and message passing can also be easier to implement correctly.

We can also draw parallels with original Paxos. For example, in a scenario in which the client finds that one of the previous register sets has the `Maybe V` decision, it picks up `V` and attempts to commit it again, which is similar to how a proposer in Paxos can propose the value after the failure of the previous proposer that was able to commit the value to at least one acceptor. Similarly, if in Paxos leader conflicts are resolved by restarting the vote with a higher proposal number, in the generalized algorithm any unwritten lower-ranked registers are set to `nil`.

Raft

Paxos was *the* consensus algorithm for over a decade, but in the distributed systems community it's been known as difficult to reason about. In 2013, a new algorithm called Raft appeared. The researchers who developed it wanted to create an algorithm that's easy to understand and implement. It was first presented in a paper titled "In Search of an Understandable Consensus Algorithm" [ONGARO14].

There's enough inherent complexity in distributed systems, and having simpler algorithms is very desirable. Along with a paper, the authors have released a reference implementation called LogCabin to resolve possible ambiguities and help future implementors to gain a better understanding.

Locally, participants store a log containing the sequence of commands executed by the state machine. Since inputs that processes receive are identical and logs contain the same commands in the same order, applying these commands to the state machine guarantees the same output. Raft simplifies consensus by making the concept of leader a first-class citizen. A leader is used to coordinate state machine manipulation and replication. There are many similarities between Raft and atomic broadcast algorithms, as well as Multi-Paxos: a single leader emerges from replicas, makes atomic decisions, and establishes the message order.

Each participant in Raft can take one of three roles:

Candidate
: Leadership is a temporary condition, and any participant can take this role. To become a leader, the node first has to transition into a candidate state, and attempt to collect a majority of votes. If a candidate neither wins nor loses the election (the vote is split between multiple candidates and none of them has a majority of votes), the new term is slated and election restarts.

Leader
: A current, temporary cluster leader that handles client requests and interacts with a replicated state machine. The leader is elected for a period called a *term*. Each term is identified by a monotonically increasing number and may continue for an arbitrary time period. A new leader is elected if the current one crashes, becomes unresponsive, or is suspected by other processes to have failed, which can happen because of network partitions and message delays.

Follower
: A passive participant that persists log entries and responds to requests from the leader and candidates. Follower in Raft is a role similar to acceptor *and* learner from Paxos. Every process begins as a follower.

To guarantee global partial ordering without relying on clock synchronization, time is divided into *terms* (also called epoch), during which the leader is unique and stable. Terms are monotonically numbered, and each command is uniquely identified by the term number and the message number within the term [HOWARD14].

It may happen that different participants disagree on which term is *current*, since they can find out about the new term at different times, or could have missed the leader election for one or multiple terms. Since each message contains a term identifier, if one of the participants discovers that its term is out-of-date, it updates the term to the higher-numbered one [ONGARO14]. This means that there *may be* several terms in flight at any given point in time, but the higher-numbered one wins in case of a conflict. A node updates the term only if it starts a new election process or finds out that its term is out-of-date.

On startup, or whenever a follower doesn't receive messages from the leader and suspects that it has crashed, it starts the leader election process. A participant attempts to become a leader by transitioning into the candidate state and collecting votes from the majority of nodes.

Figure 14-10 shows a sequence diagram representing the main components of the Raft algorithm:

Leader election
: Candidate `P1` sends a `RequestVote` message to the other processes. This message includes the candidate's term, the last term known by it, and the ID of the last log entry it has observed. After collecting a majority of votes, the candidate is successfully elected as a leader for the term. Each process can give its vote to at most one candidate.

Periodic heartbeats
: The protocol uses a heartbeat mechanism to ensure the liveness of participants. The leader periodically sends heartbeats to all followers to maintain its term. If a follower doesn't receive new heartbeats for a period called an *election timeout*, it assumes that the leader has failed and starts a new election.

Log replication / broadcast
: The leader can repeatedly append new values to the replicated log by sending `AppendEntries` messages. The message includes the leader's term, index, and term of the log entry that immediately precedes the ones it's currently sending, and *one or more* entries to store.

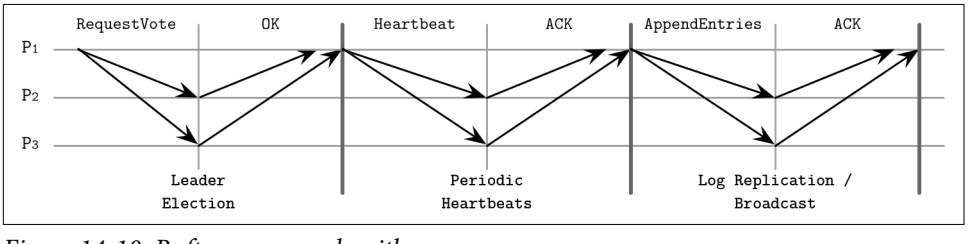

Figure 14-10. Raft consensus algorithm summary

Leader Role in Raft

A leader can be elected only from the nodes holding all committed entries: if during the election, the follower's log information is more up-to-date (in other words, has a higher term ID, or a longer log entry sequence, if terms are equal) than the candidate's, its vote is denied.

To win the vote, a candidate has to collect a majority of votes. Entries are always replicated in order, so it is always enough to compare IDs of the latest entries to understand whether or not one of the participants is up-to-date.

Once elected, the leader has to accept client requests (which can also be forwarded to it from other nodes) and replicate them to the followers. This is done by appending the entry to its log and sending it to all the followers in parallel.

When a follower receives an `AppendEntries` message, it appends the entries from the message to the local log, and acknowledges the message, letting the leader know that

it was persisted. As soon as enough replicas send their acknowledgments, the entry is considered committed and is marked correspondingly in the leader log.

Since only the most up-to-date candidates can become a leader, followers never have to bring the leader up-to-date, and log entries are only flowing from leader to follower and not vice versa.

Figure 14-11 shows this process:

- a) A new command `x = 8` is appended to the leader's log.
- b) Before the value can be committed, it has to be replicated to the majority of participants.
- c) As soon as the leader is done with replication, it commits the value locally.
- d) The commit decision is replicated to the followers.

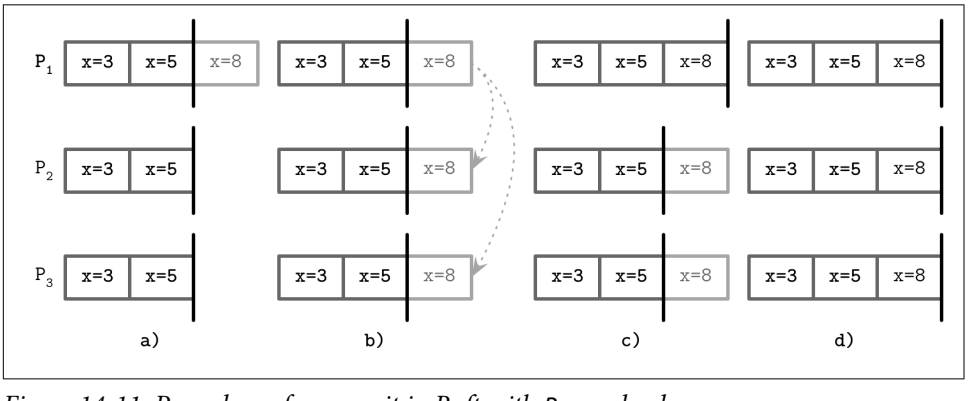

Figure 14-11. Procedure of a commit in Raft with P_1 as a leader

Figure 14-12 shows an example of a consensus round where `P1` is a leader, which has the most recent view of the events. The leader proceeds by replicating the entries to the followers, and committing them after collecting acknowledgments. Committing an entry also commits all entries preceding it in the log. Only the leader can make a decision on whether or not the entry can be committed. Each log entry is marked with a term ID (a number in the top-right corner of each log entry box) and a log index, identifying its position in the log. Committed entries are guaranteed to be replicated to the quorum of participants and are safe to be applied to the state machine in the order they appear in the log.

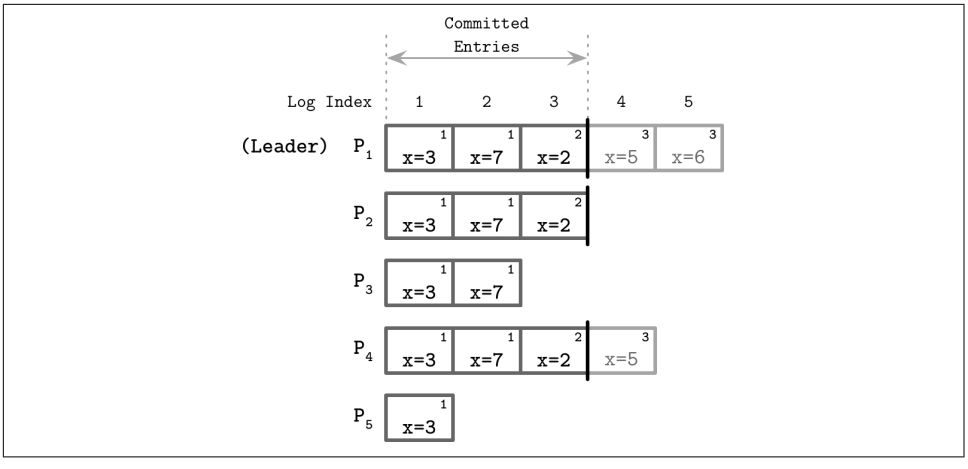

Figure 14-12. Raft state machine

Failure Scenarios

When multiple followers decide to become candidates, and no candidate can collect a majority of votes, the situation is called a *split vote*. Raft uses randomized timers to reduce the probability of multiple subsequent elections ending up in a split vote. One of the candidates can start the next election round earlier and collect enough votes, while the others sleep and give way to it. This approach speeds up the election without requiring any additional coordination between candidates.

Followers may be down or slow to respond, and the leader has to make the best effort to ensure message delivery. It can try sending messages again if it doesn't receive an acknowledgment within the expected time bounds. As a performance optimization, it can send multiple messages in parallel.

Since entries replicated by the leader are uniquely identified, repeated message delivery is guaranteed not to break the log order. Followers deduplicate messages using their sequence IDs, ensuring that double delivery has no undesired side effects.

Sequence IDs are also used to ensure the log ordering. A follower rejects a higher-numbered entry if the ID and term of the entry that immediately precedes it, sent by the leader, do not match the highest entry according to its own records. If entries in two logs on different replicas have the same term and the same index, they store the same command and all entries that precede them are the same.

Raft guarantees to never show an uncommitted message as a committed one, but, due to network or replica slowness, already committed messages can still be seen as *in progress*, which is a rather harmless property and can be worked around by retrying a client command until it is finally committed [HOWARD14].

For failure detection, the leader has to send heartbeats to the followers. This way, the leader maintains its term. When one of the nodes notices that the current leader is down, it attempts to initiate the election. The newly elected leader has to restore the state of the cluster to the last known up-to-date log entry. It does so by finding a *common ground* (the highest log entry on which both the leader and follower agree), and ordering followers to *discard* all (uncommitted) entries appended after this point. It then sends the most recent entries from its log, overwriting the followers' history. The leader's own log records are never removed or overwritten: it can only append entries to its own log.

Summing up, the Raft algorithm provides the following guarantees:

- Only one leader can be elected at a time for a given term; no two leaders can be active during the same term.
- The leader does not remove or reorder its log contents; it only appends new messages to it.
- Committed log entries are guaranteed to be present in logs for subsequent leaders and cannot get reverted, since before the entry is committed it is known to be replicated by the leader.
- All messages are identified uniquely by the message and term IDs; neither current nor subsequent leaders can reuse the same identifier for the different entry.

Since its appearance, Raft has become very popular and is currently used in many databases and other distributed systems, including CockroachDB, Etcd, and Consul. This can be attributed to its simplicity, but also may mean that Raft lives up to the promise of being a reliable consensus algorithm.

Byzantine Consensus

All the consensus algorithms we have been discussing so far assume non-Byzantine failures (see "Arbitrary Faults" on page 193). In other words, nodes execute the algorithm in "good faith" and do not try to exploit it or forge the results.

As we will see, this assumption allows achieving consensus with a smaller number of available participants and with fewer round-trips required for a commit. However, distributed systems are sometimes deployed in potentially adversarial environments, where the nodes are not controlled by the same entity, and we need algorithms that can ensure a system can function correctly even if some nodes behave erratically or even maliciously. Besides ill intentions, Byzantine failures can also be caused by bugs, misconfiguration, hardware issues, or data corruption.

Most Byzantine consensus algorithms require N^2 messages to complete an algorithm step, where N is the size of the quorum, since each node in the quorum has to com-

municate with each other. This is required to cross-validate each step against other nodes, since nodes cannot rely on each other or on the leader and have to verify other nodes' behaviors by comparing returned results with the majority responses.

We'll only discuss one Byzantine consensus algorithm here, Practical Byzantine Fault Tolerance (PBFT) [CASTRO99]. PBFT assumes independent node failures (i.e., failures can be coordinated, but the entire system cannot be taken over at once, or at least with the same exploit method). The system makes weak synchrony assumptions, like how you would expect a network to behave normally: failures may occur, but they are not indefinite and are eventually recovered from.

All communication between the nodes is encrypted, which serves to prevent message forging and network attacks. Replicas know one another's public keys to verify identities and encrypt messages. Faulty nodes may leak information from inside the system, since, even though encryption is used, every node needs to interpret message contents to react upon them. This doesn't undermine the algorithm, since it serves a different purpose.

PBFT Algorithm

For PBFT to guarantee both safety and liveness, no more than `(n - 1)/3` replicas can be faulty (where `n` is the total number of participants). For a system to sustain `f` compromised nodes, it is required to have at least `n = 3f + 1` nodes. This is the case because a majority of nodes have to agree on the value: `f` replicas might be faulty, and there might be `f` replicas that are not responding but may not be faulty (for example, due to a network partition, power failure, or maintenance). The algorithm has to be able to collect enough responses from nonfaulty replicas to *still* outnumber those from the faulty ones.

Consensus properties for PBFT are similar to those of other consensus algorithms: all nonfaulty replicas have to agree both on the set of received values and their order, despite the possible failures.

To distinguish between cluster configurations, PBFT uses *views*. In each view, one of the replicas is a *primary* and the rest of them are considered *backups*. All nodes are numbered consecutively, and the index of the primary node is `v mod N`, where `v` is the view ID, and `N` is the number of nodes in the current configuration. The view can change in cases when the primary fails. Clients execute their operations against the primary. The primary broadcasts the requests to the backups, which execute the requests and send a response back to the client. The client waits for `f + 1` replicas to respond with *the same result* for any operation to succeed.

After the primary receives a client request, protocol execution proceeds in three phases:

Pre-prepare
: The primary broadcasts a message containing a view ID, a unique monotonically increasing identifier, a payload (client request), and a payload digest. Digests are computed using a strong collision-resistant hash function, and are signed by the sender. The backup accepts the message if its view matches with the primary view and the client request hasn't been tampered with: the calculated payload digest matches the received one.

Prepare
: If the backup accepts the pre-prepare message, it enters the prepare phase and starts broadcasting `Prepare` messages, containing a view ID, message ID, and a payload digest, but without the payload itself, to all other replicas (including the primary). Replicas can move past the prepare state only if they receive `2f` prepares from *different* backups that match the message received during pre-prepare: they have to have the same view, same ID, and a digest.

Commit
: After that, the backup moves to the commit phase, where it broadcasts `Commit` messages to all other replicas and waits to collect `2f + 1` matching `Commit` messages (possibly including its own) from the other participants.

A *digest* in this case is used to reduce the message size during the prepare phase, since it's not necessary to rebroadcast an entire payload for verification, as the digest serves as a payload summary. Cryptographic hash functions are resistant to collisions: it is difficult to produce two values that have the same digest, let alone two messages with matching digests that *make sense* in the context of the system. In addition, digests are *signed* to make sure that the digest itself is coming from a trusted source.

The number `2f` is important, since the algorithm has to make sure that at least `f + 1` nonfaulty replicas respond to the client.

Figure 14-13 shows a sequence diagram of a normal-case PBFT algorithm round: the client sends a request to P_1, and nodes move between phases by collecting a sufficient number of matching responses from *properly behaving* peers. P_4 may have failed or could've responded with unmatching messages, so its responses wouldn't have been counted.

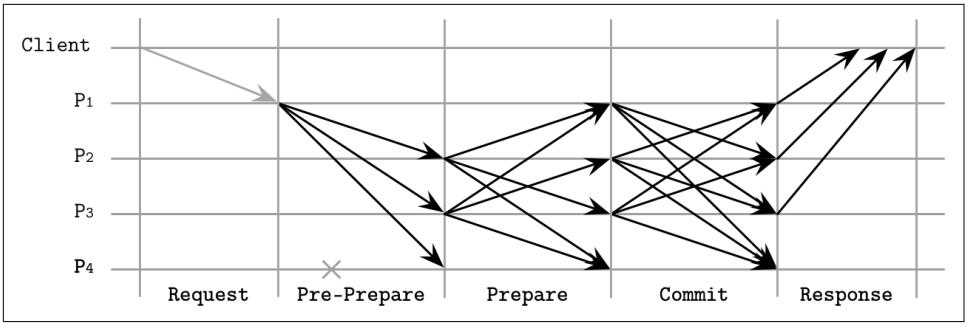

Figure 14-13. PBFT consensus, normal-case operation

During the prepare and commit phases, nodes communicate by sending messages to each other node and waiting for the messages from the corresponding number of other nodes, to check if they match and make sure that incorrect messages are not broadcasted. Peers cross-validate all messages so that only nonfaulty nodes can successfully commit messages. If a sufficient number of matching messages cannot be collected, the node doesn't move to the next step.

When replicas collect enough commit messages, they notify the client, finishing the round. The client cannot be certain about whether or not execution was fulfilled correctly until it receives `f + 1` matching responses.

View changes occur when replicas notice that the primary is inactive, and suspect that it might have failed. Nodes that detect a primary failure stop responding to further messages (apart from checkpoint and view-change related ones), broadcast a view change notification, and wait for confirmations. When the primary of the new view receives `2f` view change events, it initiates a new view.

To reduce the number of messages in the protocol, clients can collect `2f + 1` matching responses from nodes that *tentatively* execute a request (e.g., after they've collected a sufficient number of matching `Prepared` messages). If the client cannot collect enough matching tentative responses, it retries and waits for `f + 1` nontentative responses as described previously.

Read-only operations in PBFT can be done in just one round-trip. The client sends a read request to all replicas. Replicas execute the request in their tentative states, after all ongoing state changes to the read value are committed, and respond to the client. After collecting `2f + 1` responses with the same value from different replicas, the operation completes.

Recovery and Checkpointing

Replicas save accepted messages in a stable log. Every message has to be kept until it has been executed by at least `f + 1` nodes. This log can be used to get other replicas up to speed in case of a network partition, but recovering replicas need some means of verifying that the state they receive is correct, since otherwise recovery can be used as an attack vector.

To show that the state is correct, nodes compute a digest of the state for messages up to a given sequence number. Nodes can compare digests, verify state integrity, and make sure that messages they received during recovery add up to a correct final state. This process is too expensive to perform on every request.

After every `N` requests, where `N` is a configurable constant, the primary makes a *stable checkpoint*, where it broadcasts the latest sequence number of the latest request whose execution is reflected in the state, and the digest of this state. It then waits for `2f + 1` replicas to respond. These responses constitute a proof for this checkpoint, and a guarantee that replicas can safely discard state for all pre-prepare, prepare, commit, and checkpoint messages up to the given sequence number.

Byzantine fault tolerance is essential to understand and is used in storage systems deployed in potentially adversarial networks. Most of the time, it is enough to authenticate and encrypt internode communication, but when there's no trust between the parts of the system, algorithms similar to PBFT have to be employed.

Since algorithms resistant to Byzantine faults impose significant overhead in terms of the number of exchanged messages, it is important to understand their use cases. Other protocols, such as the ones described in [BAUDET19] and [BUCHMAN18], attempt to optimize the PBFT algorithm for systems with a large number of participants.

Summary

Consensus algorithms are one of the most interesting yet most complex subjects in distributed systems. Over the last few years, new algorithms and many implementations of the existing algorithms have emerged, which proves the rising importance and popularity of the subject.

In this chapter, we discussed the classic Paxos algorithm, and several variants of Paxos, each one improving its different properties:

Multi-Paxos
: Allows a proposer to retain its role and replicate multiple values instead of just one.

Fast Paxos
: Allows us to reduce a number of messages by using *fast* rounds, when acceptors can proceed with messages from proposers other than the established leader.

EPaxos
: Establishes event order by resolving dependencies between submitted messages.

Flexible Paxos
: Relaxes quorum requirements and only requires a quorum for the first phase (voting) to intersect with a quorum for the second phase (replication).

Raft simplifies the terms in which consensus is described, and makes leadership a first-class citizen in the algorithm. Raft separates log replication, leader election, and safety.

To guarantee consensus safety in adversarial environments, Byzantine fault-tolerant algorithms should be used; for example, PBFT. In PBFT, participants cross-validate one another's responses and only proceed with execution steps when there's enough nodes that obey the prescribed algorithm rules.

Further Reading

If you'd like to learn more about the concepts mentioned in this chapter, you can refer to the following sources:

Atomic broadcast
: Junqueira, Flavio P., Benjamin C. Reed, and Marco Serafini. "Zab: High-performance broadcast for primary-backup systems." 2011. In *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN '11)*: 245-256.

  Hunt, Patrick, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. "ZooKeeper: wait-free coordination for internet-scale systems." In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference (USENIX-ATC'10)*: 11.

  Oki, Brian M., and Barbara H. Liskov. 1988. "Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems." In *Proceedings of the seventh annual ACM Symposium on Principles of distributed computing (PODC '88)*: 8-17.

  Van Renesse, Robbert, Nicolas Schiper, and Fred B. Schneider. 2014. "Vive la Différence: Paxos vs. Viewstamped Replication vs. Zab."

Classic Paxos
: Lamport, Leslie. 1998. "The part-time parliament." *ACM Transactions on Computer Systems* 16, no. 2 (May): 133-169.

Lamport, Leslie. 2001. "Paxos made simple." ACM SIGACT News 32, no. 4: 51-58.

Lamport, Leslie. 2005. "Generalized Consensus and Paxos." Technical Report MSR-TR-2005-33. Microsoft Research, Mountain View, CA.

Primi, Marco. 2009. "Paxos made code: Implementing a high throughput Atomic Broadcast." (Libpaxos code: *https://bitbucket.org/sciascid/libpaxos/src/master/*.

Fast Paxos

Lamport, Leslie. 2005. "Fast Paxos." 14 July 2005. Microsoft Research.

Multi-Paxos

Chandra, Tushar D., Robert Griesemer, and Joshua Redstone. 2007. "Paxos made live: an engineering perspective." In *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing (PODC '07)*: 398-407.

Van Renesse, Robbert and Deniz Altinbuken. 2015. "Paxos Made Moderately Complex." *ACM Computing Surveys* 47, no. 3 (February): Article 42. *https://doi.org/10.1145/2673577*.

EPaxos

Moraru, Iulian, David G. Andersen, and Michael Kaminsky. 2013. "There is more consensus in Egalitarian parliaments." In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*: 358-372.

Moraru, I., D. G. Andersen, and M. Kaminsky. 2013. "A proof of correctness for Egalitarian Paxos." Technical report, Parallel Data Laboratory, Carnegie Mellon University, Aug. 2013.

Raft

Ongaro, Diego, and John Ousterhout. 2014. "In search of an understandable consensus algorithm." In *Proceedings of the 2014 USENIX conference on USENIX Annual Technical Conference (USENIX ATC'14)*, Garth Gibson and Nickolai Zeldovich (Eds.): 305-320.

Howard, H. 2014. "ARC: Analysis of Raft Consensus." Technical Report UCAM-CL-TR-857, University of Cambridge, Computer Laboratory, July 2014.

Howard, Heidi, Malte Schwarzkopf, Anil Madhavapeddy, and Jon Crowcroft. 2015. "Raft Refloated: Do We Have Consensus?" *SIGOPS Operating Systems Review* 49, no. 1 (January): 12-21. *https://doi.org/10.1145/2723872.2723876*.

Recent developments

Howard, Heidi and Richard Mortier. 2019. "A Generalised Solution to Distributed Consensus." 18 Feb 2019.