DISTRIBUTED SYSTEMS FOR PRACTITIONERS

DIMOS RAPTIS

Contents

Preface

Distributed systems are becoming ubiquitous in our life nowadays: from how we communicate with our friends to how we make online shopping and many more things. It might be transparent to us sometimes, but many companies are making use of extremely complicated software systems under the hood to satisfy our needs. By using these kind of systems, companies are capable of significant achievements, such as sending our message to a friend who is thousand miles away in a matter of milliseconds, delivering our orders despite outages of whole datacenters or searching the whole Internet by processing more than a million terabytes of data in less than a second. Putting all of this into perspective, it's easy to understand the value that distributed systems bring in the current world and why it's useful for software engineers to be able to understand and make use of distributed systems.

However, as easy and fascinating as it might seem, the area of distributed systems is a rather complicated one with many different execution models and failure modes. As a result, in order for one to simply understand how to use a 3rd party library or verify the correctness of a distributed system under construction, one has to digest a vast amount of information first. Distributed systems have been a really hot academic topic for the last decades and tremendous progress has been achieved, albeit through a large number of papers with one building on top of the previous ones usually. This sets a rather high barrier to entry for newcomers and practitioners that just want to understand the basic building blocks, so that they can be confident in the systems they are building without any aspirations of inventing new algorithms or protocols.

The ultimate goal of this book is to help these people get started with distributed systems in an easy and intuitive way. It was born out of my initiation to the topic, which included a lot of transitions between excitement, confusion and enlightenment.

Of course, it would be infeasible to tackle all the existing problems in the space of distributed computing. So, this book will focus on:

- establishing the basic principles around distributed systems
- explaining *what is* and *what is not* possible to achieve
- explaining the basic algorithms and protocols, by giving easy-to-follow examples and diagrams
- explaining the thinking behind some design decisions
- expanding on how these can be used in practice and what are some of the issues that might arise when doing so
- eliminating confusion around some terms (i.e. *consistency*) and foster thinking about trade-offs when designing distributed systems
- providing plenty of additional resources for people that are willing to invest more time in order to get a deeper understanding of the theoretical parts

Who is this book for

This book is aimed at software engineers that have some experience in building software systems and have no or some experience in distributed systems. We assume no knowledge around concepts and algorithms for distributed systems. This book attempts to gradually introduce the terms and explain the basic algorithms in the simplest way possible, providing many diagrams and examples. As a result, this book can also be useful to people that don't develop software, but want to get an introduction to the field of distributed systems. However, this book does not aim to provide a full analysis or proof of every single algorithm. Instead, the book aims to help the reader get the intuition behind a concept or an algorithm, while also providing the necessary references to the original papers, so that the reader can study other parts of interest in more depth.

Acknowledgements

As any other book, this book might have been written by a single person, but that would not have been possible without the contribution of many others. As a result, credits should be given to all my previous employers and colleagues that have given me the opportunity to work with large-scale, distributed systems and appreciate both their capabilities and complexities and the distributed systems community that was always open to answer any questions. I would also like to thank Richard Gendal Brown for reviewing the case study on Corda and giving feedback that was very useful in helping me to add clarity and remove ambiguity. Of course, this book would not have been possible without the understanding and support of my partner in life, Maria.

Part I

Fundamental Concepts

Chapter 1

Introduction

What is a distributed system and why we need it

First of all, we need to define what a distributed system is. Multiple, different definitions can be found, but we will use the following:

> "A distributed system is a system whose components are ***located on different networked computers***, which communicate and coordinate their actions by ***passing messages*** to one another."[1]

As shown in Figure 1.1, this network can either consist of direct connections between the components of the distributed system or there could be more components that form the backbone of the network (if communication is done through the Internet for example). These components can take many forms; they could be servers, routers, web browsers or even mobile devices. In an effort to keep an abstract and generic view, in the context of this book we'll refer to them as **nodes**, being agnostic to their real form. In some cases, such as when providing a concrete example, it might be useful to escape this generic view and see how things work in real-life. In these cases, we might explain in detail the role of each node in the system.

As we will see later, the 2 parts that were highlighted in the definition above are central to how distributed systems function:

- the various parts that compose a distributed system are located remotely, separated by a network.

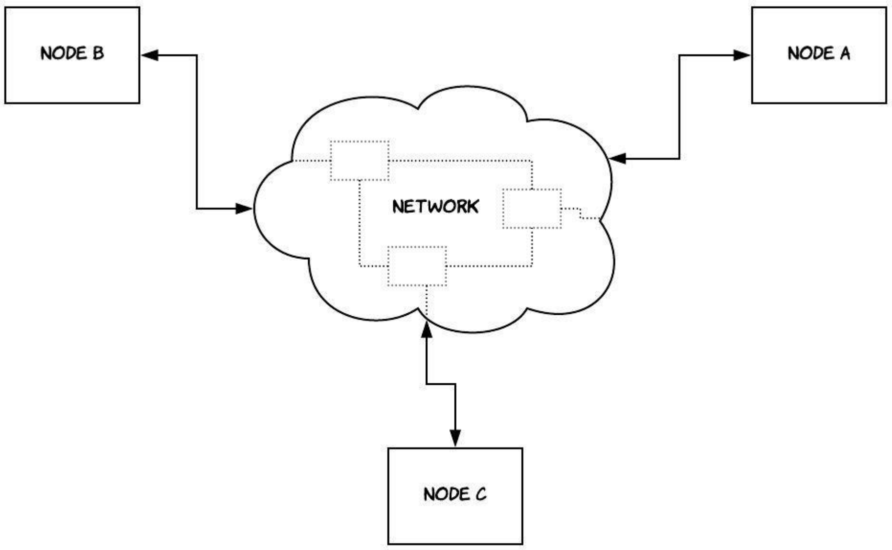


Figure 1.1: A distributed system

- the main mechanism of communication between them is by exchanging messages, using this network that separates them.

Now that we have defined what a distributed system is, let's explore its value.

Why do we really need distributed systems ?

Looking at all the complexity that distributed systems introduce, as we will see during this book, that's a valid question. The main benefits of distributed systems come mostly in the following 3 areas:

- performance
- scalability
- availability

Let's explain each one separately. The performance of a single computer has certain limits imposed by physical constraints on the hardware. Not only that, but after a point, improving the hardware of a single computer in order to achieve better performance becomes extremely expensive. As a result, one can achieve the same performance with 2 or more low-spec computers as with a single, high-end computer. **So, distributed systems allow us to achieve better performance at a lower cost**. Note that better performance can translate to different things depending on the context, such as lower latency per request, higher throughput etc.

> "Scalability is the capability of a system, network, or process to handle a growing amount of work, or its potential to be enlarged to accommodate that growth." [2]

Most of the value derived from software systems in the real world comes from storing and processing data. As the customer base of a system grows, the system needs to handle larger amounts of traffic and store larger amounts of data. However, a system composed of a single computer can only scale up to a certain point, as explained previously. **Building a distributed system allows us to split and store the data in multiple computers, while also distributing the processing work amongst them**[1]. As a result of this, we are capable of scaling our systems to sizes that would not even be imaginable with a single-computer system.

In the context of software systems, availability is the probability that a system will work as required when required during the period of a mission. Note that nowadays most of the online services are required to operate all the time (known also as 24/7 service), which makes this a huge challenge. So, when a service states that it has 5 nines of availability, this means that it operates normally for 99.999% of the time. This implies that it's allowed to be down for up to 5 minutes a year, to satisfy this guarantee. Thinking about how unreliable hardware can be, one can easily understand how big an undertaking this is. Of course, using a single computer, it would be infeasible to provide this kind of guarantees. **One of the mechanisms that are widely used to achieve higher availability is redundancy, which means storing data into multiple, redundant computers**. So, when one of them fails, we can easily and quickly switch to another one, preventing our customers from experiencing this failure. Given that data are stored now in multiple computers, we end up with a distributed system!

Leveraging a distributed system we can get all of the above benefits. However, as we will see later on, there is a tension between them and several other

[1]The approach of scaling a system by adding resources (memory, CPU, disk) to a single node is also referred to as *vertical scaling*, while the approach of scaling by adding more nodes to the system is referred to as *horizontal scaling*.

properties. So, in most of the cases we have to make a trade-off. To do this, we need to understand the basic constraints and limitations of distributed systems, which is the goal of the first part of this book.

The fallacies of distributed computing

Distributed systems are subject to many more constraints, when compared to software systems that run in a single computer. As a result, developing software for distributed systems is also very different. However, people that are new to distributed systems make assumptions, based on their experience developing software for systems that run on a single computer. Of course, this creates a lot of problems down the road for the systems they build. In an effort to eliminate this confusion and help people build better systems, L Peter Deutsch and others at Sun Microsystems created a collection of these false assumptions, which is now known as the **fallacies of distributed computing**[2]. These are the following:

1. The network is reliable.[3][4]
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

We will focus on those that are mostly relevant to this book here: 1, 2 and 3. The first fallacy is sometimes enforced by abstractions provided to developers from various technologies and protocols. Even though protocols, like TCP, can make us believe that network is reliable and never fails, this is just an illusion. We should understand that network connections are also built on top of hardware that will also fail at some point and we should design our systems accordingly. The second assumption is also enforced nowadays by libraries, which attempt to model remote procedure calls as local calls,

[2]See: https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

such as gRPC[3] or Thrift[4]. We should always keep in mind that there is a difference of several orders of magnitude in latency between a call to a remote system and a local memory access (from milliseconds to nanoseconds). This is getting even worse, when we are talking about calls between datacenters in different continents, so this is another thing to keep in mind when deciding about how we want to geo-distribute our system. The third one is getting weaker nowadays, since there have been significant improvements in the bandwidth that can be achieved during the last decades. Still, even though we can build high-bandwidth connections in our own datacenter, this does not mean that we will be able to use all of it, if our traffic needs to cross the Internet. This is an important consideration to keep in mind, when making decisions about the topology of our distributed system and when requests will have to travel through the Internet.

There's one more fallacy that's not included in the above set, but it's still very common amongst people new to distributed systems and can also create a lot of confusion. If we were to follow the same style as above, we would probably phrase it in the following way:

> "Distributed systems have a global clock, which can be used to identify when events happen"

This assumption can be quite deceiving, since it's somewhat intuitive and holds true when working in systems that are not distributed. For instance, an application that runs in a single computer can use the computer's local clock in order to decide when events happen and what's the order between them. Nonetheless, that's not true in a distributed system, where every node in the system has its own local clock, which runs at a different rate from the other ones. There are ways to try and keep the clocks in sync, but some of them are very expensive and do not eliminate these differences completely. This limitation is again bound by physical laws[5]. An example of such an approach is the TrueTime API that was built by Google [5], which exposes explicitly the clock uncertainty as a first-class citizen. However, as we will see in the next chapters of the book, when one is mainly interested in cause and effects, there are other ways to reason about time using logical clocks

[3]See: https://grpc.io/
[4]See: https://thrift.apache.org/
[5]See: https://en.wikipedia.org/wiki/Time_dilation

instead.

Why distributed systems are hard

In general, distributed systems are hard to design, build and reason about, thus increasing the risk of error. This will become more evident later in the book while exploring some algorithms that solve fundamental problems that emerge in distributed systems. It's worth questioning: why are distributed systems so hard? The answer to this question can help us understand what are the main properties that make distributed systems challenging, thus eliminating our blind spots and providing some guidance on what are some of the aspects we should be paying attention to.

The main properties of distributed systems that make them challenging to reason about are the following:

- network asynchrony
- partial failures
- concurrency

Network asynchrony is a property of communication networks that cannot provide strong guarantees around delivery of events, e.g. a maximum amount of time required for a message to be delivered. This can create a lot of counter-intuitive behaviours that would not be present in non-distributed systems. For instance, this is in contrast to memory operations that can provide much stricter guarantees[6]. For instance, in a distributed system messages might take extremely long to be delivered or they might be delivered out of order.

Partial failures are cases where only some components of a distributed system fail. This behaviour can come in contrast to certain kind of applications deployed in a single server that work under the assumption that either the whole server has crashed or everything is working fine. It introduces significant complexity when there is a requirement for atomicity across components in a distributed system, i.e. we need to ensure that an operation is either applied to all the nodes of a system or to none of them. The chapter about distributed transactions analyses this problem.

Concurrency is execution of multiple computations happening at the same

[6]See: https://en.wikipedia.org/wiki/CAS_latency

time and potentially on the same piece of data interleaved with each other. This introduces additional complexity, since these different computations can interfere with each other and create unexpected behaviours. This is again in contrast to simplistic applications with no concurrency, where the program is expected to run in the order defined by the sequence of commands in the source code. The various types of problematic behaviours that can arise from concurrency are explained in the chapter that talks about isolation later in the book.

As explained, these 3 properties are the major contributors of complexity in the field of distributed systems. As a result, it will be useful to keep them in mind during the rest of the book and when building distributed systems in real life so that you can anticipate edge cases and handle them appropriately.

Correctness in distributed systems

The correctness of a system can be defined in terms of the properties it must satisfy. These properties can be of the following types:

- Safety properties
- Liveness properties

A safety property defines something that must never happen in a correct system, while a liveness property defines something that must eventually happen in a correct system. As an example, considering the correctness properties of an oven, we could say that the property of "the oven not exceeding a maximum temperature threshold" is a safety property. The property of "the oven eventually reaching the temperature we specified via the button" is a liveness property. Similar to this example, in distributed systems, it's usually more important to make sure that the system satisfies the safety properties than the liveness ones. Throughout this book, it will become clear that there is an inherent tension between safety and liveness properties. Actually, as we will see later in the book, there are some problems, where it's physically impossible to satisfy both kinds of properties, so compromises are made for some liveness properties in order to maintain safety.

System models

Real-life distributed systems can differ drastically in many dimensions, depending on the network where they are deployed, the hardware they are running on etc. Thus, we need a common framework so that we can solve problems in a generic way without having to repeat the reasoning for all the different variations of these systems. In order to do this, we can create a model of a distributed system by defining several properties that it must satisfy. Then, if we prove an algorithm is correct for this model, we can be sure that it will also be correct for all the systems that satisfy these properties.

The main properties that are of interest in a distributed system have to do with:

- how the various nodes of a distributed system interact with each other
- how a node of a distributed system can fail

Depending on the nature of communication, we have 2 main categories of systems: **synchronous** and **asynchronous** systems. A *synchronous* system is one, where each node has an accurate clock and there is a known upper bound on message transmission delay and processing time. As a result, the execution is split into rounds so that every node can send a message to another node, the messages are delivered and every node computes based on the messages just received, all nodes running in lock-step. An *asynchronous* system is one, where there is no fixed upper bound on how long it takes for a message to be delivered or how much time elapses between consecutive steps of a node. The nodes of the system do not have a common notion of time and thus run in independent rates. The challenges arising from network asynchrony have already been discussed previously. So, it should be clear by now that the first model is much easier to describe, program and reason about. However, the second model is closer to real-life distributed systems, such as the Internet, where we cannot have control over all the components involved and there are very limited guarantees on the time it will take for a message to be sent between two places. As a result, most of the algorithms we will be looking at this book assume an asynchronous system model.

There are also several different types of failure. The most basic categories are:

- **Fail-stop**: A node halts and remains halted permanently. Other nodes can detect that the node has failed (i.e. by communicating with it).

- **Crash**: A node halts and remains halted, but it halts in a silent way. So, other nodes may not be able to detect this state (i.e. they can only assume it has failed on the basis of not being able to communicate with it).
- **Omission**: A node fails to respond to incoming requests.
- **Byzantine**: A node exhibits arbitrary behavior: it may transmit arbitrary messages at arbitrary times, it may stop or take an incorrect step.

Byzantine failures can be exhibited, when a node does not behave according to the specified protocol/algorithm, i.e. because the node has been compromised by a malicious actor or because of a software bug. Coping with these failures introduces significant complexity to the resulting solutions. At the same time, most distributed systems in companies are deployed in environments that are assumed to be private and secure. Fail-stop failures are the simplest and the most convenient ones from the perspective of someone that builds distributed systems. However, they are also not very realistic, since there are cases in real-life systems where it's not easy to identify whether another node has crashed or not. As a result, most of the algorithms analysed in this book work under the assumption of crash failures.

The tale of exactly-once semantics

As described in the beginning of the book, the various nodes of a distributed system communicate with each other by exchanging messages. Given that the network is not reliable, these messages might get lost. Of course, to cope with this, nodes can retry sending them hoping that the network will recover at some point and deliver the message. However, this means that messages might be delivered multiple times, as shown in Figure 1.2, since the sender can't know what really happened.

This duplicate delivery of a message can create disastrous side-effects. For instance, think what would happen if that message is supposed to signal transfer of money between 2 bank accounts as part of a purchase; a customer might be charged twice for a product. To handle scenarios like this, there are multiple approaches to ensure that the processing of a message will only be done once, even though it might be delivered multiple times.

One approach is using *idempotent operations.* Idempotent is an operation that can be applied multiple times without changing the result beyond the initial

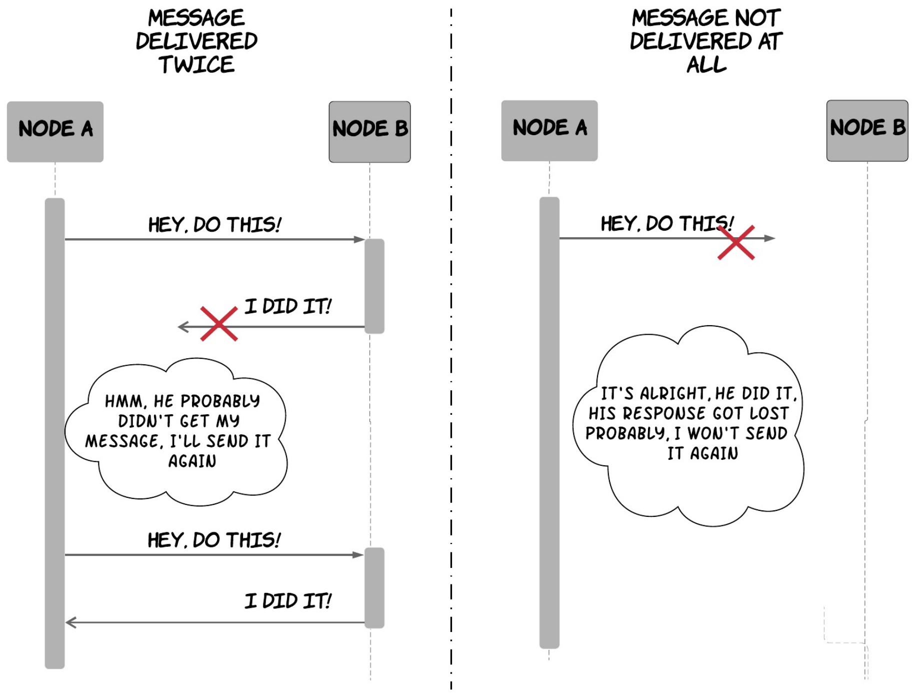

MESSAGE DELIVERED TWICE
NODE A
NODE B
HEY, DO THIS!
I DID IT!
HMM, HE PROBABLY DIDN'T GET MY MESSAGE, I'LL SEND IT AGAIN
HEY, DO THIS!
I DID IT!
MESSAGE NOT DELIVERED AT ALL
NODE A
NODE B
HEY, DO THIS!
IT'S ALRIGHT, HE DID IT, HIS RESPONSE GOT LOST PROBABLY, I WON'T SEND IT AGAIN

Figure 1.2: Intricacies of a non-reliable network in distributed systems

application. An example of an idempotent operation is adding a value in a set of values. Even if this operation is applied multiple times, the applications that follow the first one will have no effect, since the value will already have been added in the set. Of course, this is under the assumption that other operations cannot remove values from the set. Otherwise, the retried operation might add a value that had been removed in the meanwhile. On the contrary, an example of a non-idempotent operation would be increasing a counter by one, which has additional side-effects every time it's applied. By making use of idempotent operations, we can have a guarantee that even if a message is delivered multiple times and the operation is repeated, the end result will be the same.

However, as demonstrated previously idempotent operations commonly impose tight constraints on the system. So, in many cases we cannot build our system, so that all operations are idempotent by nature. In these cases, we can use a *de-duplication approach*, where we give every message a unique identifier and every retried message contains the same identifier as the original. In this way, the recipient can remember the set of identifiers it has received and executed already and avoid executing operations that have already been executed. It is important to note that in order to do this, one must have control on both sides of the system (sender and receiver). This is due to the fact that the ID generation is done on the sender side, but the deduplication process is done on the receiver side. As an example, imagine a scenario where an application is sending emails as part of an operation. Sending an e-mail is not an idempotent operation, so if the e-mail protocol does not support de-duplication on the receiver side, then we cannot be absolutely sure that every e-mail is shown exactly once to the recipient.

When thinking about exactly-once semantics, it's useful to distinguish between the notions of delivery and processing. In the context of this discussion, let's consider delivery being the arrival of the message at the destination node at the hardware level. Then, we consider processing being the handling of this message from the software application layer of the node. In most cases, what we really care about is how many times a message is processed, not how many times it has been delivered. For instance, in our previous e-mail example, we are mainly interested in whether the application will display the same e-mail twice, not whether it will receive it twice. As the previous examples demonstrated, **it's impossible to have exactly-once delivery** in a distributed system. It's still *sometimes* **possible** though **to have exactly-once processing**. With all that said, it's important to understand the difference between these 2 notions and make clear what you are referring to, when you are talking about exactly-once semantics.

Also, as a last note, it's easy to see that at-most-once delivery semantics and at-least-once delivery semantics can be trivially implemented. The former can be achieved by sending every message only one time no matter what happens, while the latter one can be achieved by sending a message continuously, until we get an acknowledgement from the recipient.

Failure in the world of distributed systems

It is also useful to understand that it is very difficult to identify failure because of all the characteristics of a distributed system described so far. The asynchronous nature of the network in a distributed system can make it very hard to differentiate between a node that has crashed and a node that is just really slow to respond to requests. The main mechanism used to detect failures in a distributed systems are **timeouts**. Since messages can get infinitely delayed in an asynchronous network, timeouts impose an artificial upper bound on these delays. As a result, when a node is slower than this bound, we can assume that the node has failed. This is useful, since otherwise the system might be blocked eternally waiting for nodes that have crashed under the assumption that they might just be extremely slow.

However, this timeout does not represent an actual limit, so it creates the following trade-off. Selecting a smaller value for this timeout means that our system will waste less time waiting for nodes that have crashed. At the same time, the system might be declaring dead some nodes that have not crashed, but they are just being a bit slower than expected. On the other hand, selecting a larger value for this timeout means that the system will be more lenient with slow nodes. However, it also implies that the system will be slower in identifying crashed nodes, thus wasting time waiting for them in some cases. This is illustrated in Figure 1.3.

In fact, this is a very important problem in the field of distributed systems. The component of a node that is used to identify other nodes that have failed is called a **failure detector**. As we explained previously, this component is very important for various algorithms that need to make progress in the presence of failures. There has been extensive research about failure detectors [6]. The different categories of failure detectors are distinguished by 2 basic properties that reflect the aforementioned trade-off: completeness and accuracy. Completeness corresponds to the percentage of crashed nodes a failure

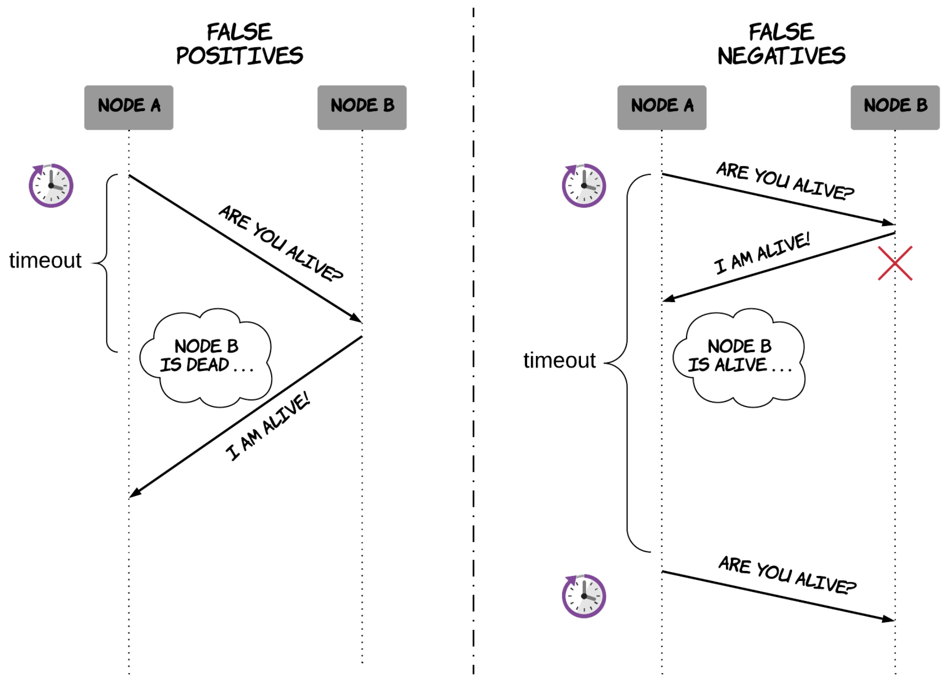


Figure 1.3: Trade-offs in failure detection

detector succeeded in identifying in a certain period. Accuracy corresponds to the number of mistakes a failure detector made in a certain period. A perfect failure detector is one that is characterised by the strongest form of completeness and accuracy, namely one that can successfully detect every faulty process without ever thinking a node has crashed before it actually crashes. As expected, it is impossible to build a perfect failure detector in purely asynchronous systems. Still, even imperfect failure detectors can be used to solve difficult problems, such as the problem of consensus which is described later.

Stateful and Stateless systems

We could say that a system can belong in one of the 2 following categories:

- stateless systems
- stateful systems

A stateless system is one that maintains no state of what has happened in the past and is capable of performing its capabilities, purely based on the inputs provided to it. For instance, a contrived stateless system is one that receives a set of numbers as input, calculates the maximum of them and returns it as the result. Note that these inputs can be direct or indirect. Direct inputs are those included in the request, while indirect inputs are those potentially received from other systems to fullfil the request. For instance, imagine a service that calculates the price for a specific product by retrieving the initial price for it and any currently available discounts from some other services and then performing the necessary calculations with this data. This service would still be stateless. On the other hand, stateful systems are responsible for maintaining and mutating some state and their results depend on this state. As an example, imagine a system that stores the age of all the employees of a company and can be asked for the employee with the maximum age. This system is stateful, since the result depends on the employees we've registered so far in the system.

There are some interesting observations to be made about these 2 types of systems:

- Stateful systems can be really useful in real-life, since computers are much more capable in storing and processing data than humans.

- Maintaining state comes with additional complexity, such as deciding what's the most efficient way to store it and process it, how to perform back-ups etc.
- As a result, it's usually wise to create an architecture that contains clear boundaries between stateful components (which are performing business capabilities) and stateless components (which are responsible for handling data).
- Last and most relevant to this book, it's much easier to design, build and scale distributed systems that are stateless when compared to stateful ones. The main reason for this is that all the nodes (e.g. servers) of a stateless system are considered to be identical. This makes it a lot easier to balance traffic between them and scale by adding or removing servers. However, stateful systems present many more challenges, since different nodes can hold different pieces of data, thus requiring additional work to direct traffic to the right place and ensure each instance is in sync with the other ones.

As a result, some of the book's examples might include stateless systems, but most of the problems we will cover in this book are present mostly in stateful systems.

Chapter 2

Basic concepts and theorems

Partitioning

As we described previously, one of the major benefits of distributed systems is scalability, allowing us to store and process datasets much larger than what one could do with a single machine. One of the primary mechanisms of achieving scalability, is called partitioning. *Partitioning* is the process of splitting a dataset into multiple, smaller datasets and then assigning the responsibility of storing and processing them to different nodes of a distributed system. This allows us to increase the size of the data our system can handle, by adding more nodes to the system.

There are 2 different variations of partitioning: *vertical partitioning* and *horizontal partitioning* (also called sharding). The terms vertical and horizontal originate from the era of relational databases, which established the notion of a tabular view of data.[1] In this view, data consist of rows and columns, where a row is a different entry in the dataset and each column is a different attribute for every entry. Figure 2.1 contains a visual depiction of the difference between these 2 approaches.

Vertical partitioning involves splitting a table into multiple tables with fewer columns and using additional tables to store columns that serve the purpose of relating rows across tables (commonly referred to as a `join` operation). These different tables can then be stored in different nodes. Normalization[2]

[1] See: https://en.wikipedia.org/wiki/Relational_model
[2] See: https://en.wikipedia.org/wiki/Database_normalization

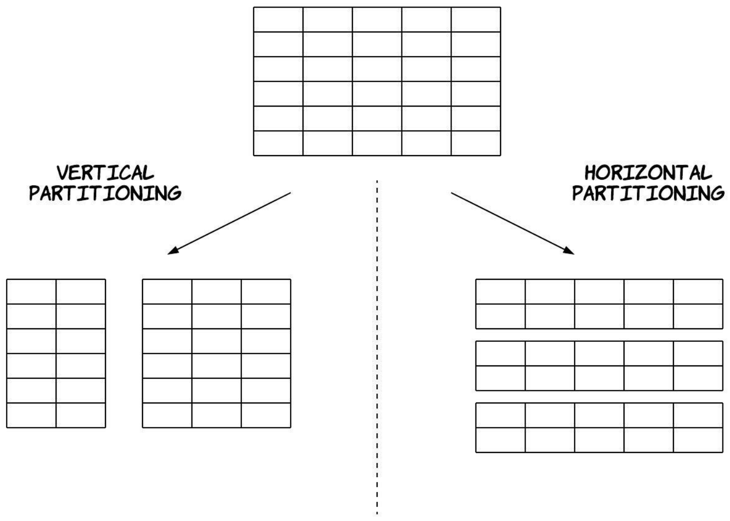


Figure 2.1: Vertical and Horizontal Partitioning

is one way to perform vertical partitioning, but general vertical partitioning can go far beyond that, splitting columns, even when they are normalized.

On the other hand, horizontal partitioning involves splitting a table into multiple, smaller tables, where each of those tables contain a percentage of the rows of the initial table. These different sub-tables can then be stored in different nodes. There are multiple strategies for performing this split, as we will see later on. A simplistic approach is an alphabetical split. For instance, in a table containing the students of a school, we could partition horizontally, using the surname of the students, as shown in Figure 2.2.

Partitioning helps with allowing a system to handle larger datasets more efficiently, but it also introduces some limitations. In a vertically partitioned system, requests that need to combine data from different tables (i.e. `join` operations) become less efficient, because these requests might now have to access data from multiple nodes. In a horizontally partitioned system, this is usually avoided, because all the data for each row is located in the same node. However, it can happen for requests that are searching for a range of rows and these rows belong to multiple nodes. Another important implication of horizontal partitioning is the potential for loss of transactional semantics. When storing data in a single machine, it's easy to perform multiple operations in an atomic way, so that either all of them succeed or

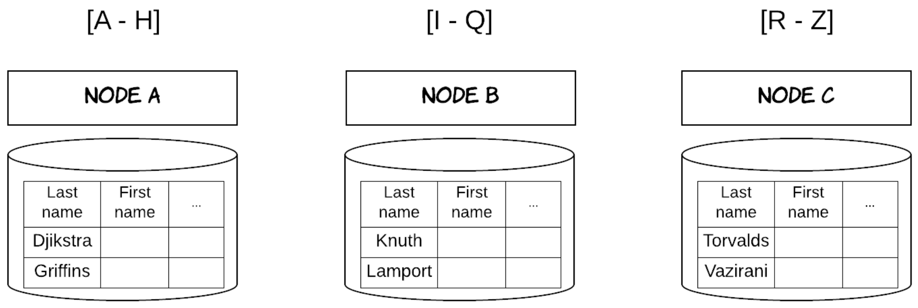


Figure 2.2: Horizontal Partitioning using alphabetical order

none of them succeeds, but this is much harder to achieve in a distributed system (as we will see in the chapter about distributed transactions). As a result, when partitioning data horizontally, it's much harder to perform atomic operations over data that reside in different nodes. This is a common theme in distributed systems; there's no silver bullet, one has to make trade-offs in order to achieve a desired property.

Vertical partitioning is mainly a data modelling practice, which can be performed by the engineers designing a system, sometimes independently of the storage systems that will be used. However, horizontal partitioning is commonly provided as a feature of distributed databases, so it's important for engineers to know how it works under the hood in order to make proper use of these systems. As a result, we will focus mostly on horizontal partitioning in this book.

Algorithms for horizontal partitioning

There are a lot of different algorithms for performing horizontal partitioning. In this section, we will study some of these algorithms, discussing the advantages and drawbacks of each one.

Range partitioning is a technique, where a dataset is split into ranges, according to the value of a specific attribute. Each range is then stored in a separate node. The case we described previously with the alphabetical split is an example of range partitioning. Of course, the system should store and maintain a list of all these ranges, along with a mapping, indicating which node stores a specific range. In this way, when the system is receiving a request for a specific value (or a range of values), it consults this mapping to identify to which node (or nodes, respectively) the request should be redirected.

The advantages of this technique are:

- its simplicity and ease of implementation.
- the ability to perform range queries, using the value that is used as the partitioning key.
- a good performance for range queries using the partitioning key, when the queried range is small and resides in a single node.
- easy and efficient way to adjust the ranges (re-partition), since one range can be increased or decreased, exchanging data only between 2 nodes.

Some of its disadvantages are:

- the inability to perform range queries, using other keys than the partitioning key.
- a bad performance for range queries using the partitioning key, when the queried range is big and resides in multiple nodes.
- an uneven distribution of the traffic or the data, causing some nodes to be overloaded. For example, some letters are more frequent as initial letters in surnames,[3] which means that some nodes might have to store more data and process more requests.

Some systems that leverage a range partitioning technique are Google's BigTable [7] and Apache HBase.[4]

Hash partitioning is a technique, where a hash function is applied to a specific attribute of each row, resulting in a number that determines which partition (and thus node) this row belongs to. For the sake of simplicity, let's assume we have one partition per node (as in the previous example) and a hash function that returns an integer. If we have `n` number of nodes in our system and trying to identify which node a student record with a surname `s` is located at, then we could calculate it using the formula `hash(s) mod n`. This mapping process needs to be done both when writing a new record and when receiving a request to find a record for a specific value of this attribute.

[3]See: http://surnamestudies.org.uk/teaching/micro.htm
[4]See: https://hbase.apache.org/

The advantages of this technique are:

- the ability to calculate the partitioning mapping at runtime, without needing to store and maintain the mapping. This has benefits both in terms of data storage needs and performance (since no additional request is needed to find the mapping).
- a bigger chance of the hash function distributing the data more uniformly across the nodes of our system, thus preventing some nodes from being overloaded.

Some disadvantages of this technique are:

- the inability to perform range queries at all (even for the attribute used as a partitioning key), without storing additional data or querying all the nodes.
- adding/removing nodes from the systems causes re-partitioning, which results in significant movement of data across all nodes of the system.

Consistent hashing is a partitioning technique, having very similar characteristics to the previous one, but solving the problem of increased data movement during re-partitioning. The way it works is the following: each node in the system is randomly assigned an integer in a range `[0, L]`, called *ring* (i.e. `[0, 360]`). Then, a record with a value `s` for the attribute used as partitioning key is located to the node that is the next one after the point `hash(s) mod L` in the ring. As a result, when a new node is added to the ring, it receives data only from the previous node in the ring, without any more data needed to be exchanged between any other nodes. In the same way, when a node is removed from the ring, its data just need to be transferred to the next node in the ring. For a visual representation of this behaviour and the difference between these 2 different algorithms, see Figure 2.3.

Some disadvantages of this technique are:

- the potential for non-uniform distribution of the data, because of the random assignment of the nodes in the ring.
- the potential for creating more imbalanced data distribution as nodes are added or removed. For example, when a node is removed, its dataset is not distributed evenly across the system, but it's transferred to a single node.

Both of these issues can be mitigated by using the concept of *"virtual nodes"*,

HASH PARTITIONING

Partitioning key	Hash	Chosen server (N = 4)	Chosen server (n = 3)	Migrated
Djikstra	134013486	2	0	YES
Griffins	392184030	2	0	YES
Knuth	549203819	3	2	YES
Lamport	692048502	2	0	YES
Torvalds	549382205	1	2	YES
Vazirani	474960373	1	1	NO

CONSISTENT HASHING

Partitioning key	Hash	mod L (L = 360)	Chosen server {N1,N2,N3,N4}	Chosen server {N1, N2, N4}	Migrated
Djikstra	134013486	246	4	4	NO
Griffins	392184030	30	2	2	NO
Knuth	549203819	59	2	2	NO
Lamport	692048502	342	1	1	NO
Torvalds	549382205	245	4	4	NO
Vazirani	474960373	133	3	4	YES

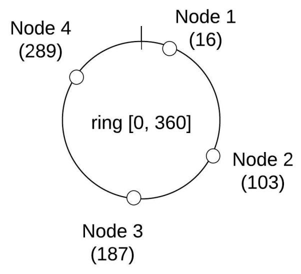


Figure 2.3: Re-partitioning, when a node (N3) is removed, in hash partitioning and consistent hashing

where each physical node is assigned multiple locations (virtual nodes) in the ring. For more discussion on this concept, feel free to read the Dynamo paper [8]. Another widely used system that makes use of consistent hashing is Apache Cassandra [9].

Replication

As we discussed in the previous section, partitioning can improve the scalability and performance of a system, by distributing data and request load to multiple nodes. However, the introduction mentioned another dimension that benefits from the usage of a distributed system and that was availability. This property directly translates to the ability of the system to remain functional despite failures in parts of it. *Replication* is the main technique used in distributed systems in order to increase availability. It consists of storing the same piece of data in multiple nodes (called *replicas*), so that if one of them crashes, data is not lost and requests can be served from the other nodes in the meanwhile.

However, the benefit of increased availability from replication comes with a set of additional complications. Replication implies that the system now has multiple copies of every piece of data, which must be maintained and kept in sync with each other on every update. Ideally, replication should function transparently to the end-user (or engineer), creating the illusion that there is only a single copy of every piece of data. This makes a distributed system look like a simple, centralised system of a single node, which is much easier to reason about and develop software around.

Of course, this is not always possible; it might require significant hardware resources or giving up other desired properties to achieve this ideal. For instance, engineers are sometimes willing to accept a system that provides much higher performance, giving occasionally a non-consistent view of the data as long as this is done only under specific conditions and in a specific way they can account for, when designing the overall application. As a result of this, there are 2 main strategies for replication:

- **Pessimistic replication**: this strategy tries to guarantee from the beginning that all of the replicas are identical to each other, as if there was only a single copy of the data all along.
- **Optimistic replication** (also called *lazy replication*): this strategy allows the different replicas to diverge, guaranteeing that they will

converge again if the system does not receive any updates (also known as *quiesced*) for a period of time.

Replication is a very active field in research, so there are many different algorithms. As an introduction, we will now discuss the 2 main techniques: *single-master replication* and *multi-master replication.*

Single-master replication

Single-master replication is a technique, where a single node amongst the replicas is designated as *master* (or *primary*) and is responsible for receiving all the updates[5]. The remaining replicas are commonly referred to as *slaves* (or *secondaries*) and they can only handle read requests. Every time the master receives an update, it's responsible for propagating this update to the other nodes besides executing it locally, ensuring all the replicas will maintain a consistent view of the data. This propagation of the updates can be done in 2 ways: either *synchronously* or *asynchronously.*

In **synchronous replication**, the node can reply to the client indicating the update has been completed, only after having received acknowledgements from the other replicas that they have also performed the update on their local storage. This guarantees that after an update has been acknowledged to a client, the client will be able to view this update in a subsequent read, no matter which replica it reads from. Furthermore, it provides increased durability, since the update will not be lost, even if the master crashes right after acknowledging the update. However, this technique can make write requests slower, since the master has to wait until responses have been received from all the replicas.

In **asynchronous replication**, the node can reply to the client as soon as it has performed the update in its local storage, without waiting for responses from the other replicas. This increases performance significantly for write requests, since the client does not pay the penalty of the network requests to the other replicas anymore. However, this comes at a cost of reduced consistency and decreased durability. After a client has received a response for an update request, he might read older (stale) values in a subsequent read, if this operation happens in one of the replicas that has not performed the update yet. On top of that, if the master node crashes right

[5]This technique is also known as *primary-backup* replication.

after acknowledging an update and the "propagation" requests to the other replicas are lost, then an update that has been acknowledged is eventually lost. The difference between these 2 techniques is visualised in Figure 2.4.

The main advantages of single-master replication are:

- it's simple to understand and to implement.
- concurrent operations are serialized in the master node, obviating the need for more complicated, distributed concurrency protocols. In general, this property also makes it easier to support transactional operations.
- it's quite scalable for workloads that are read-heavy, since capacity for read requests can be increased, by adding more read replicas.

Its main disadvantages are:

- it's not very scalable for write-heavy workloads, since the capacity for writes is determined by the capacity of a single node (the master).
- it imposes an obvious trade-off between performance, durability and consistency.
- failing over to a slave node, when the master node crashes is not instant, it might create some downtime and it also introduces risk of errors. In general, there are two different approaches for performing the failover: manual or automated. In the manual approach, the operator selects the new master node and instructs all the nodes accordingly. This is the safest approach, but it can also incur a significant downtime. The alternative is an automated approach, where slave nodes detect that the master node has crashed (e.g. via periodic heartbeats) and attempt to elect a new master node. This can be faster, but it's also quite risky, because there are many different ways in which the nodes can get confused and arrive to an incorrect state. The chapter about consensus will be covering in more detail this topic, called leader election.
- even though read capacity can be scaled by adding more slave nodes, the network bandwidth of the master node can end up being a bottleneck, if there's a big number of slaves listening for updates. An interesting variation of single-master replication that mitigates this problem is a technique, called chain replication, where nodes form a chain, propagating the updates linearly [10].

Most of the widely used databases, such as PostgreSQL[6] or MySQL,[7] use

[6]See: https://www.postgresql.org/

[7]See: https://www.mysql.com/

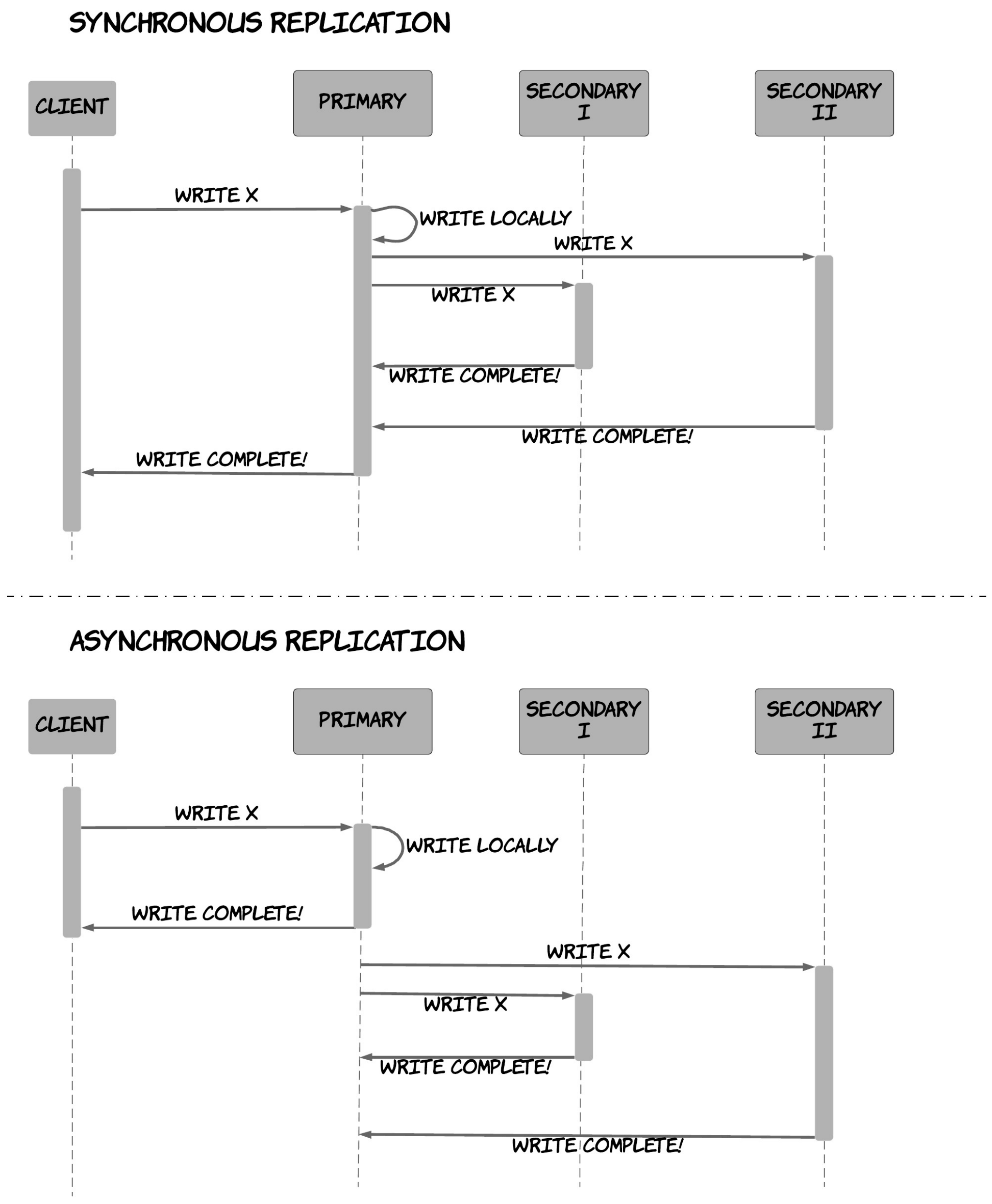


Figure 2.4: Synchronous vs asynchronous replication

a single-master replication technique, supporting both asynchronous and synchronous replication.

Multi-master replication

As we've seen in the previous section, the single-master replication is a technique, which is easy to implement and operate, it can easily support transactions and can hide the distributed nature of the underlying system (i.e. when using synchronous replication). However, it has some limitations in terms of performance, scalability and availability. As we've already discussed, there are some kinds of applications, where availability and performance is much more important than data consistency or transactional semantics. A frequently cited example is that of an e-commerce shopping cart, where the most important thing is for the customers to be able to access their cart at all times and be able to add items in a quick and easy way. Compromising consistency to achieve this is acceptable, as long as there is data reconciliation at some point. For instance, if 2 replicas diverge because of intermittent failures, the customer can still resolve any conflicts, during the checkout process.

Multi-master replication is an alternative replication technique that favors higher availability and performance over data consistency[8]. In this technique, all replicas are considered to be equal and can accept write requests, being also responsible for propagating the data modifications to the rest of the group. There is a significant difference with the single-master replication; in multi-master replication, there is no single, master node that serializes the requests imposing a single order, since write requests are concurrently handled by all the nodes. This means that nodes might disagree on what the right order is for some requests. This is usually referred to as a *conflict*. In order for the system to remain operational when this happens, the nodes need to resolve this conflict, agreeing on a single order amongst the available ones. Figure 2.5 shows an example, where 2 write requests can potentially result in a conflict, depending on the latency of the propagation requests between the nodes of the system. In the first diagram, write requests are processed in the same order in all the nodes, so there is no conflict. In the second diagram, the write requests are processed in different order in the various nodes (because of network delays), which results in a conflict. In this

[8]This technique is also known as *multi-primary* replication.

case, a subsequent read request could receive different results, depending on the node that handles the request, unless we resolve the conflict so that all the nodes converge again to a single value.

There are many different ways to resolve conflicts, depending on the guarantees the system wants to provide. An important characteristic of different approaches to resolving conflicts is whether they resolve the conflict eagerly or lazily. In the first case, the conflict is resolved during the write operation. In the second case, the write operation proceeds maintaining multiple, alternative versions of the data record and these are eventually resolved to a single version later on, i.e. during a subsequent read operation. For instance, some common approaches for conflict resolution are:

- exposing conflict resolution to the clients. In this approach, when there is a conflict, the multiple available versions are returned to the client, who selects the right version and returns it to the system, resolving the conflict. An example of this could be the shopping cart application, where the customer selects the correct version of his/her cart.
- last-write-wins conflict resolution. In this approach, each node in the system tags each version with a timestamp, using a local clock. During a conflict, the version with the latest timestamp is selected. Since there can't be a global notion of time, as we've discussed, this technique can lead to some unexpected behaviours, such as write A overriding write B, even though B happened "as a result" of A.
- conflict resolution using causality tracking algorithms. In this approach, the system makes use of an algorithm that keeps track of causal relationships between different requests. When there is a conflict between 2 writes (A, B) and one is determined to be the cause of the other one (suppose A is the cause of B), then the resulting write (B) is retained. However, keep in mind that there can still be writes that are not causally related (requests that are actually concurrent), where the system cannot make an easy decision.

We'll elaborate more on some of these approaches later in the chapter about time and order.

Quorums in distributed systems

The main pattern we've seen so far is writes being performed to all the replica nodes, while reads are performed to one of them. Ensuring writes

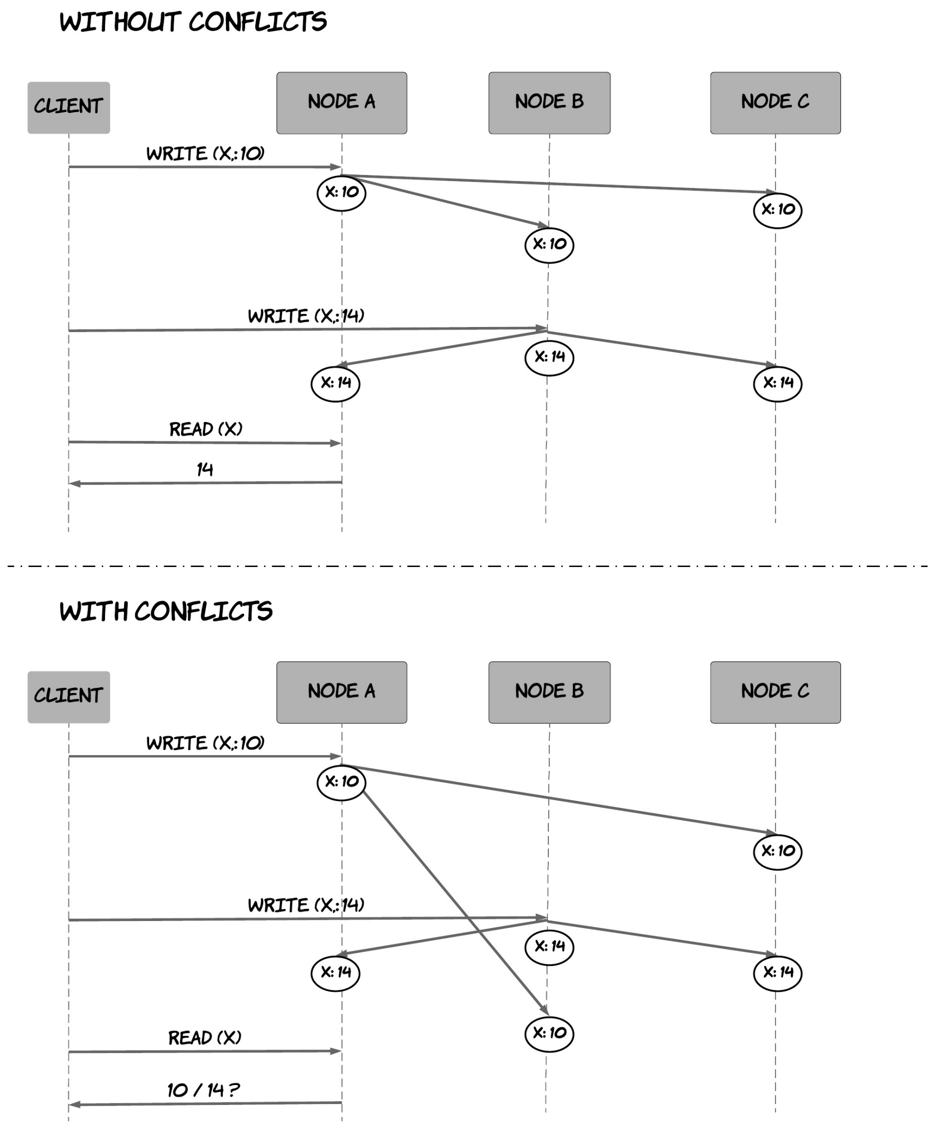


Figure 2.5: Conflicts in multi-master replication

are performed to all of them (synchronously) before replying to the client, we can guarantee that the subsequent reads will have seen all the previous writes regardless of the node that processes the read operation. However, this means that availability is quite low for write operations, since failure of a single node makes the system unable to process writes, until the node has recovered. Of course, the reverse strategy could be used; writing data only to the node that is responsible for processing a write operation, but processing read operations, by reading from all the nodes and returning the latest value. This would increase significantly the availability of writes, but it would decrease the availability of reads at the same time.

A useful mechanism in achieving a balance in this trade-off is using *quorums*. For instance, in a system of 3 replicas, we could say that writes need to complete in 2 nodes (also known as a quorum of 2), while reads need to retrieve data from 2 nodes. In this way, we could be sure that reads will read the latest value, because at least one of the nodes in the read quorum will be included in the latest write quorum as well. This is based on the fact that in a set of 3 elements, 2 subsets of 2 elements must have at least 1 common element.

This technique was introduced in a past paper [11] as a quorum-based voting protocol for replica control. In general, in a system that has a total of V replicas, every read operation should obtain a read quorum of V_r replicas, while a write operation should obtain a write quorum of V_w replicas, respectively. The values of these quorums should obey the following properties:

- $V_r + V_w > V / 2$
- $V_w > V / 2$

The first rule ensures that a data item is not read and written by 2 operations concurrently, as we just described. The second rule ensures that there is at least one node that will receive both 2 write operations and can impose an order on them. Essentially, this means 2 write operations from 2 different operations cannot occur concurrently on the same data item. Both of the rules together guarantee that the associated distributed database behaves as a centralized, one-replica database system. What this means exactly will become more clear in the sections that follow, which provide more formal definitions of various properties of distributed systems.

The concept of a quorum is really useful in distributed systems that are composed of multiple nodes. As we will see later in the book, it has been used extensively in other areas, like distributed transactions or consensus protocols. The concept is intentionally introduced early on in the book, so that it is easier to identify it as a pattern in the following chapters.

Safety guarantees in distributed systems

Since distributed systems involve a lot of complexity, some safety guarantees are used to ensure that the system will behave in specific expected ways. This makes it easier for people to reason about a system and any potential anomalies that can occur, so that they can build proper safeguards to prevent these anomalies from happening. The main safety guarantees that systems provide are around the following two properties:

- atomicity
- isolation
- consistency

The concepts of atomicity and isolation originate from database research and ACID transactions, while by consistency in this book we will mostly refer to the notion of consistency made popular by the CAP theorem. Thus, before going any further it would be useful to have a look at these topics first.

It is interesting to observe that each one of these safety guarantees is tightly related to one of the aforementioned reasons distributed systems are hard. Achieving atomicity is in a distributed system is challenging because of the possibility of partial failures. Achieving consistency is challenging because of the network asynchrony and achieving isolation is also challenging because of the inherent concurrency of distributed systems.

ACID transactions

ACID is a set of properties of database transactions that are used to provide guarantees around the expected behaviour of transactions in the event of errors, power failures etc. More specifically, these properties are:

- **Atomicity** (A): this property guarantees that a transaction composed of multiple operations is treated as a single unit. This means that either all operations of the transaction are executed or none of them is. This concept of atomicity translates to distributed systems, where the system might need to execute the same operation in multiple nodes of the system in an atomic way, so that the operation is either executed to all the nodes or to none. This topic will be covered more extensively in the chapter about distributed transactions.

- **Consistency** (C): this property guarantees that a transaction can only transition the database from one valid state to another valid state, maintaining any database invariants. However, these invariants are application-specific and defined by every application accordingly. For example, if an application has a table A with records that refer to records in a table B through a foreign key relationship[9], the database will prevent a transaction from deleting a record from table A, unless any records in table B referenced from this record have already been deleted. Note that this is not the concept of consistency we will be referring to in the context of distributed systems, that concept will be presented below.
- **Isolation** (I): this property guarantees that even though transactions might be running concurrently and have data dependencies, the end result will be as if one of them was executing at a time, so that there was no interference between them. This prevents a large number of anomalies that will be discussed later.
- **Durability** (D): this property guarantees that once a transaction has been committed, it will remain committed even in the case of failure. In the context of single-node, centralised systems, this usually means that completed transactions (and their effects) are recorded in non-volatile storage. In the context of distributed systems, this might mean that transactions need to be durably stored in multiple nodes, so that recovery is possible even in the presence of total failures of a node along with its storage facilities.

The CAP Theorem

The CAP Theorem [12] is one of the most fundamental theorems in the field of distributed systems, outlining an inherent trade-off in the design of distributed systems. It states that it's impossible for a distributed data store to simultaneously provide more than 2 of the following properties:

- **Consistency**[10]: this means that every successful read request will

[9]See: https://en.wikipedia.org/wiki/Foreign_key

[10]As implied earlier, the concept of consistency in the CAP theorem is completely different from the concept of consistency in ACID transactions. The notion of consistency

receive the result of the most recent write request.

- **Availability**: this means that every request receives a non-error response, without any guarantees on whether that reflects the most recent write request.
- **Partition Tolerance**: this means that the system can continue to operate despite an arbitrary number of messages being dropped by the network between nodes due to a network partition.

It is very important to understand though that partition tolerance is not a property you can abandon. In a distributed system, there is always the risk of a network partition. If this happens, then the system needs to make a decision either to continue operating while compromising data consistency or stop operating, thus compromising availability. However, there is no such thing as trading off partition tolerance in order to maintain both consistency and availability. As a result, what this theorem really states is the following:

> "In the presence of a partition, a distributed system can be either consistent or available."

Let's attempt to schematically prove this theorem in a simplistic way. As shown in Figure 2.6, let's imagine a distributed system consisting of 2 nodes. This distributed system can act as a plain register, holding the value of a variable, called `X`. Now, let's assume that at some point there is a network failure between the 2 nodes of the system, resulting in a network partition between them. A user of the system is performing a write and then a read - it could also be 2 different users performing the operations. We will examine the case where each operation is processed by a different node of the system. In that case, the system has 2 options: it can either fail one of the operations (breaking the availability property) or it can process both of the operations returning a stale value from the read (breaking the consistency property). It cannot process both of the operations successfully, while also ensuring that the read returns the latest value, which is the one written by the write operation. The reason is that the results of the write cannot be propagated from node A to node B due to the network partition.

This theorem is really important, because it has helped establish this basic limitation that all distributed systems are imposed to. This forced designers of distributed systems to make explicit trade-offs between availability and consistency and engineers become aware about these properties and choose the right system appropriately. When looking at the literature or reading

as presented in the CAP theorem is the one that is more important for distributed systems.

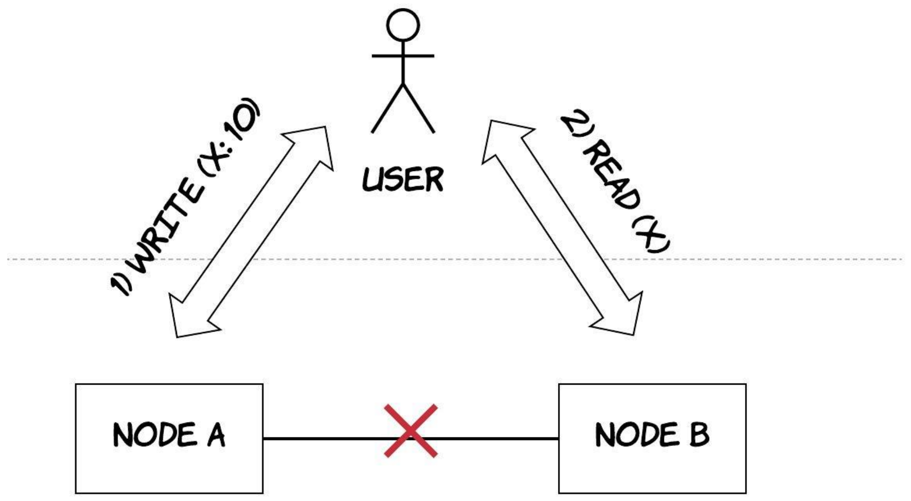


Figure 2.6: Handling a network partition in a distributed system

documentation of distributed systems, you will notice systems are usually classified in 2 basic categories, CP and AP, depending on which property the system violates during a network partition. Sometimes, you might even find a third category, called CA. As explained previously, there is no such category for distributed systems and people usually refer either to one of the other two categories instead or to a non-distributed system, such as a single node database.

There is another important thing to note about the CAP theorem: this choice between consistency and availability needs to be made only during a network partition. At all other times, both of these properties can be satisfied. However, even during normal operation when no network partition is present, there's a different trade-off between latency and consistency. In order for the system to guarantee data consistency, it will have to essentially delay write operations until the data have been propagated across the system successfully, thus taking a latency hit. An example of this trade-off is the single-master replication scheme we previously described. In this setting, a synchronous replication approach would favor consistency over latency, while asynchronous replication would benefit from reduced latency at the cost of consistency.

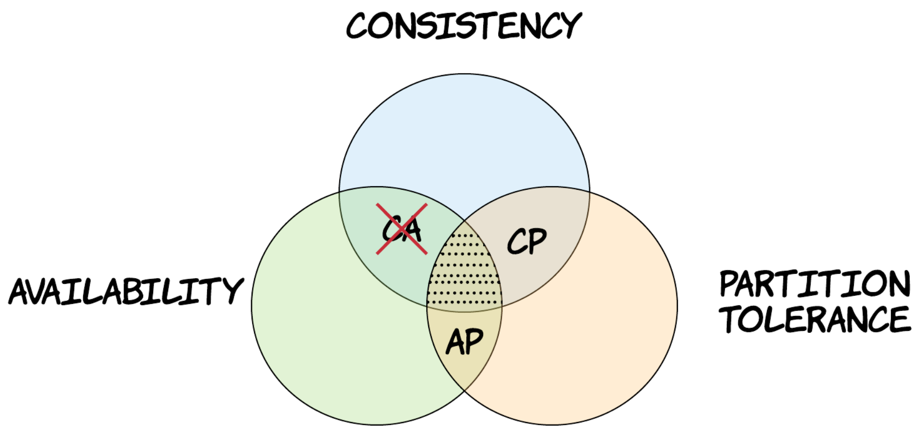


Figure 2.7: Categories of distributed systems according to the CAP theorem

There is actually an extension to the CAP theorem, called the PACELC theorem, captured in a separate article [13]. This theorem states that:

- in the case of a network partition (P), the system has to choose between availability (A) and consistency (C)
- but else (E) when the system is operating normally in the absence of network partitions, the system has to choose between latency (L) and consistency (C).

As a result, each branch of this theorem creates 2 sub-categories of systems. The first part of the theorem defines the two categories we have already seen: AP and CP. The second part defines two new categories: EL and EC. These sub-categories are combined to form 4 categories in total: AP/EL, CP/EL, AP/EC, CP/EC. For instance, a system from the AP/EL category will prioritise availability during a network partition and it will prioritise latency during normal operation. In most of the cases, systems are designed with an overarching principle in mind, which is usually either performance and availability or data consistency. As a result, most of the systems tend to fall into the categories AP/EL or CP/EC. However, there are still systems that cannot be strictly classified in one of these categories, since they have various levers that can be used to tune the system differently when needed. Still, this theorem serves as a good indicator of the various forces at play in a distributed system. You can find a table with the categorisation of several distributed systems along these dimensions in the associated Wikipedia page[11].

Consistency models

In the previous section, we defined *consistency* as the property that every successful read request will return the result of the most recent write. In fact, this was an oversimplification, since there are many different forms of consistency. In this section, we will introduce the forms that are the most relevant to the topics of this book.

As with many other things, in order to define what each of these forms really is, one needs to build a formal model. This is usually called a **consistency model** and it defines the set of execution histories[12] that are valid in a system amongst all the possible ones. In layman's terms, a model defines formally what behaviours are possible in a distributed system. Consistency models are extremely useful, because they help us formalise the behaviour of a system. Systems can then provide guarantees about their behaviour and software engineers can be confident that the way they use a distributed system (i.e. a distributed database) will not violate any safety properties they care about. In essence, software engineers can treat a distributed system as a black box that provides a set of properties, while remaining unaware of all the complexity that the system assumes internally in order to provide these. We say that a consistency model A is stronger than model B, when the first one allows fewer histories. Alternatively, we could say model A makes more assumptions or poses more restrictions on the possible behaviours of the system. Usually, the stronger the consistency model a system satisfies the easier it is to build an application on top of it, since the developer can rely on stricter guarantees.

There are many different consistency models in the literature. In the context of this book, we will focus on the most fundamental ones, which are the following:

- Linearizability
- Sequential Consistency

[11]See: https://en.wikipedia.org/wiki/PACELC_theorem

[12]A *history* is a collection of operations, including their concurrent structure (i.e. the order they are interleaved during execution).

- Causal Consistency
- Eventual Consistency

A system that supports the consistency model of **linearizability**[14] is one, where operations appear to be instantaneous to the external client. This means that they happen at a specific point from the point the client invokes the operation to the point the client receives the acknowledgement by the system the operation has been completed. Furthermore, once an operation is complete and the acknowledgement has been delivered to the client, it is visible to all other clients. This implies that if a client C2 invokes a read operation after a client C1 has received the completion of its write operation, then C2 should see the result of this (or a subsequent) write operation. This property of operations being "instantaneous" and "visible" after they are completed seems obvious, right ? However, as we have discussed previously, *there is no such thing as instantaneity in a distributed system.* Figure 2.8 might help you understand why. When thinking about a distributed system as a single node, it seems obvious that every operation happens at a specific instant of time and it's immediately visible to everyone. However, when thinking about the distributed system as a set of cooperating nodes, then it becomes clear that this should not be taken for granted. For instance, the system in the bottom diagram is not linearizable, since $T_4 > T_3$, but still the second client won't observe the read, because it hasn't propagated to the node that processes the read operation yet. To relate this to some of the techniques and principles we've discussed previously, the non-linearizability comes from the use of asynchronous replication. By using a synchronous replication technique, we could make the system linearizable. However, that would mean that the first write operation would have to take longer, until the new value has propagated to the rest of the nodes (remember the latency-consistency trade-off from the PACELC theorem!). As a result, one can realise that linearizability is a very powerful consistency model, which can help us treat complex distributed systems as much simpler, single-node datastores and reason about our applications more efficiently. Moreover, leveraging atomic instructions provided by hardware (such as CAS operations[13]), one can build more sophisticated logic on top of distributed systems, such as mutexes, semaphores, counters etc., which would not be possible under weaker consistency models.

Sequential Consistency is a weaker consistency model, where operations are allowed to take effect before their invocation or after their completion.

[13]See: https://en.wikipedia.org/wiki/Compare-and-swap

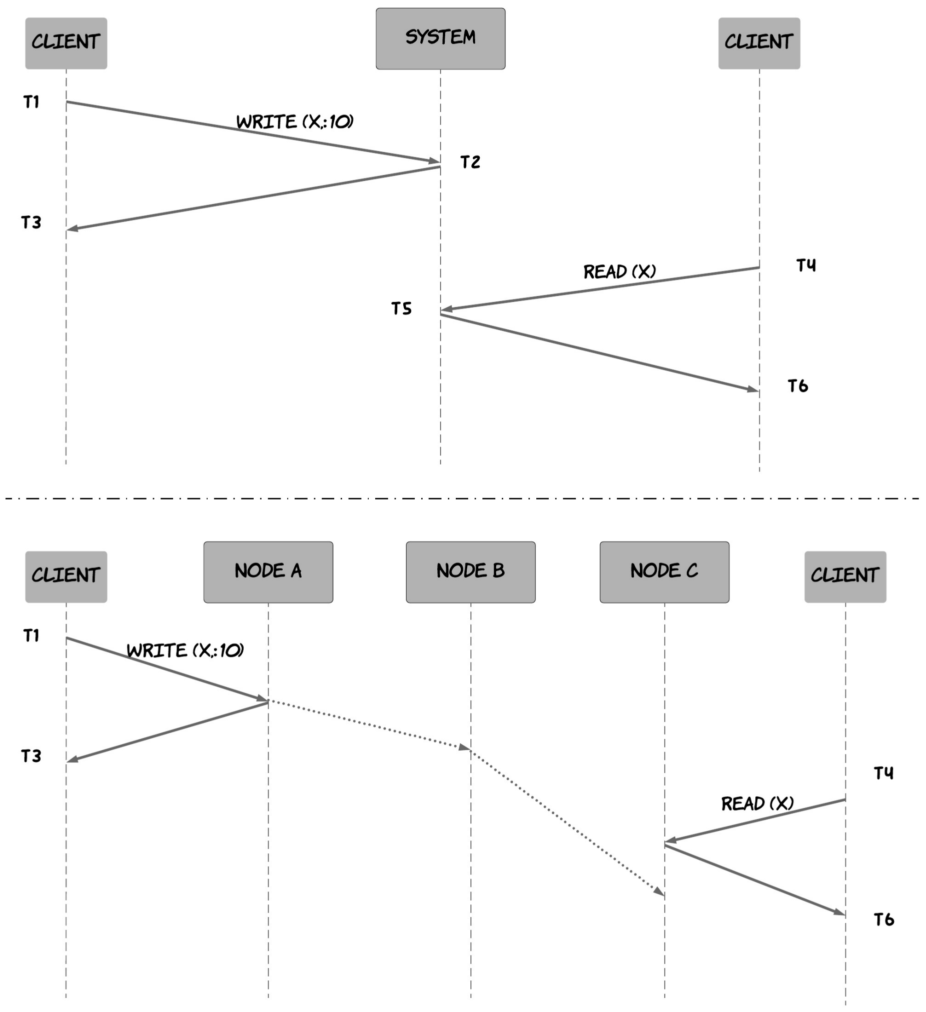


Figure 2.8: Why linearizability is not obvious in a distributed system

As a result, it provides no real-time guarantees. However, operations from different clients have to be seen in the same order by all other clients *and* operations of every single client preserve the order specified by its program (in this "global" order). This allows many more histories than linearizability, but still poses some constraints that can be useful to real-life applications. For example, in a social networking application, one usually does not really care what's the ordering of posts between some of his/her friends, but there's still an expectation that posts from a single friend are displayed in the right order (the one he/she published them at). Following the same logic, one expects his/her comments in a post to appear in the order that he/she submitted them. These are all properties that are captured by this model.

In some cases, we don't even need to preserve this ordering specified by each client's program, as long as *causally related* operations are displayed in the right order. In our previous example, one could accept comments from one of his/her friends being displayed in a different order than the one he/she originally submitted them, as long as every comment is displayed after the comment it replies to. This would be expected, since there is a *cause-and-effect*[14] relationship between a comment and the comments that constitute replies to it. This is the **causal consistency** model, which requires that only operations that are causally related need to be seen in the same order by all the nodes. Thus, unlike sequential consistency, the other operations that are not causally related can be seen in different orders in the various clients of the system, also without the need to maintain the order of each client's program. Of course, in order to achieve that each operation needs to contain some information signalling whether it depends on other operations or not. This does not need to be related to time at all and it can be an application-specific property, as in the example we previously described. Causal consistency is one of the weaker forms of consistency, while still preventing a common class of unintuitive behaviours.

There are still even simpler applications that do not have the notion of a cause-and-effect and they would benefit from an even simpler consistency model. For instance, it could be acceptable that the order of operations can be different between the multiple clients of the system and reads do not need to return the latest write, as long as the system *eventually* arrives at a stable state. In this state, if no more write operations are performed, read operations will return the same result. This is the model of **eventual consistency**. It is one of the weakest forms of consistency, since it does not

[14]See: https://en.wikipedia.org/wiki/Causality

really provide any guarantees around the perceived order of operations or the final state the system converges to. It can still be a useful model for some applications, which do not require stronger assumptions or can detect and resolve inconsistencies at the application level.

Note that there are many more consistency models, besides the ones we explained here.[15]

When explaining the CAP theorem, we encountered the term *consistency*, but which of all ?

The C property in the CAP theorem refers to the linearizability model we previously described. This means it's impossible to build a system that will be available during a network partition, while also being linearizable. In fact, there has been research that shows that even some weaker forms of consistency, such as sequential consistency, cannot be supported in tandem with availability under a network partition [15].

This vast number of different consistency models creates a significant amount of complexity. As we explained previously, modelling consistency is supposed to help us reason about these systems. However, the explosion of consistency models can have the opposite effect. The CAP theorem can conceptually draw a line between all these consistency models and separate them into 2 major categories: strong consistency models and weak consistency models. Strong consistency models correspond to the C in the CAP theorem and cannot be supported in systems that need to be available during network partitions. On the other hand, weak consistency models are the ones that can be supported, while also preserving availability during a network partition.

Looking at the guarantees provided by several popular distributed systems nowadays (i.e. Apache Cassandra, DynamoDB etc.), there are 2 models that are commonly supported. The first one is strong consistency, specifically linearizability. The second one is weak consistency, specifically eventual consistency. Most probably, the reasons most of the systems converged to these 2 models are the following:

- Linearizability was selected amongst the available strong consistency models, because in order to support a strong consistency model, a system needs to give up availability, as part of the CAP theorem. It then seems reasonable to provide the strongest model amongst the

[15]See: https://en.wikipedia.org/wiki/Consistency_model

available ones, facilitating the work of the software engineers having to work with it.
- Eventual Consistency was selected amongst the available weak consistency models thanks to its simplicity and performance. Thinking along the same lines, given the application relinquishes the strict guarantees of strong consistency for increased performance, it might as well accept the weakest guarantees possible to get the biggest performance boost it can. This makes it much easier for people designing and building applications on top of distributed systems to make a decision, when deciding which side of the CAP theorem they prefer to build their application on.

Isolation levels

As mentioned already, the inherent concurrency in distributed systems creates the potential for anomalies and unexpected behaviours. Specifically, transactions that are composed of multiple operations and run concurrently can lead to different results depending on how their operations are interleaved. As a result, there is still a need for some formal models that define what is possible and what is not in the behaviour of a system.

These are called **isolation levels**. We will study the most common ones here which are the following:

- **Serializability**
- **Repeatable read**
- **Snapshot Isolation**
- **Read Committed**
- **Read Uncommitted**

Unlike the consistency models presented in the previous section, some of these isolation levels do not define *what is possible* via some formal specification. Instead, they define *what is not possible*, i.e. which anomalies are prevented amongst the ones that are already known. Of course, stronger isolation levels prevent more anomalies at the cost of performance. Let's first have a look at the possible anomalies before examining the various levels.

The origin of the isolation levels above and the associated anomalies was essentially the ANSI SQL-92 standard[16]. However, the definitions in this standard were ambiguous and missed some possible anomalies. Subsequent research[17] examines more anomalies extensively and attempts a stricter definition of these levels. The basic parts will be covered in this section, but feel free to refer to it if you are looking for a deeper analysis.

The anomalies covered here are the following:

- Dirty writes
- Dirty reads
- (Fuzzy) non-repeatable reads
- Phantom reads
- Lost updates
- Read skew
- Write skew

A **dirty write** occurs when a transaction overwrites a value that has previously been written by another transaction that is still in-flight and has not been committed yet. One reason dirty writes are problematic is they can violate integrity constraints. For example, imagine there are 2 transactions A and B, where transaction A is running the operations `[x=1, y=1]` and transaction B is running the operations `[x=2, y=2]`. Then, a serial execution of them would always result in a situation where `x` and `y` have the same value, but in a concurrent execution where dirty writes are possible this is not necessarily true. An example could be the following execution `[x=1, x=2, y=2, commit B, y=1, commit A]` that would result in `x=2` and `y=1`. Another problem of dirty writes is they make it impossible for the system to automatically rollback to a previous image of the database. As a result, this is an anomaly that needs to be prevented in most cases.

A **dirty read** occurs when a transaction reads a value that has been written by another transaction that has not been committed yet. This is problematic, since decisions might be made by the system depending on these values even though the associated transactions might be rolled back subsequently. Even in the case where these transactions eventually commit, this can still be problematic though. An example is the classic scenario of a bank transfer where the total amount of money should be observed to be the same at all times. However, if a transaction A is able to read the balance of two accounts that are involved in a transfer right in the middle of another transaction B that performs the transfer from account 1 to account 2, then it will look like as if some money have been lost from account 1. However, there are a few cases where allowing dirty reads can be useful, if done with care. One such case is to generate a big aggregate report on a full table, when one can tolerate some inaccuracies on the numbers of the report. It can also be useful when troubleshooting an issue and one wants to inspect the state of the database in the middle of an ongoing transaction.

A **fuzzy** or **non-repeatable read** occurs when a value is retrieved twice during a transaction (without it being updated in the same transaction) and the value is different. This can lead to problematic situations similar to the example presented above for dirty reads. Other cases where this can lead to problems is if the first read of the value is used for some conditional logic and the second is used in order to update data. In this case, the transaction might be acting on stale data.

A **phantom read** occurs when a transaction does a predicate-based read and another transaction writes or removes a data item matched by that predicate while the first transaction is still in flight. If that happens, then the first transaction might be acting again on stale data or inconsistent data. For example, let's say transaction A is running 2 queries to calculate the maximum and the average age of a specific set of employees. However, between the 2 queries transaction B is interleaved and inserts a lot of old employees in this set, thus making transaction A return an average that is larger than the maximum! Allowing phantom reads can be safe for an application that is not making use of predicate-based reads, i.e. performing only reads that select records using a primary key.

A **lost update** occurs when two transactions read the same value and then try to update it to two different values. The end result is that one of the two updates survives, but the process executing the other update is not informed that its update did not take effect, thus called *lost update*. For instance, imagine a warehouse with various controllers that are used to update the database when new items arrive. The transactions are rather simple, reading the number of items currently in the warehouse, adding the number of new items to this number and then storing the result back to the database. This anomaly could lead to the following problem: transactions A and B read simultaneously the current inventory size (say 100 items), add the number of new items to this (say 5 and 10 respectively) and then store this back to the database. Let's assume that transaction B was the last one to write, this means that the final inventory is 110, instead of 115. Thus, 5 new items were not recorded! See Figure 2.9 for a visualisation of this example. Depending on the application, it might be safe to allow lost updates in some cases. For example, consider an application that allows multiple administrators to update specific parts of an internal website used by employees of a company.

In this case, lost updates might not be that catastrophic, since employees can detect any inaccuracies and inform the administrators to correct them without any serious consequences.

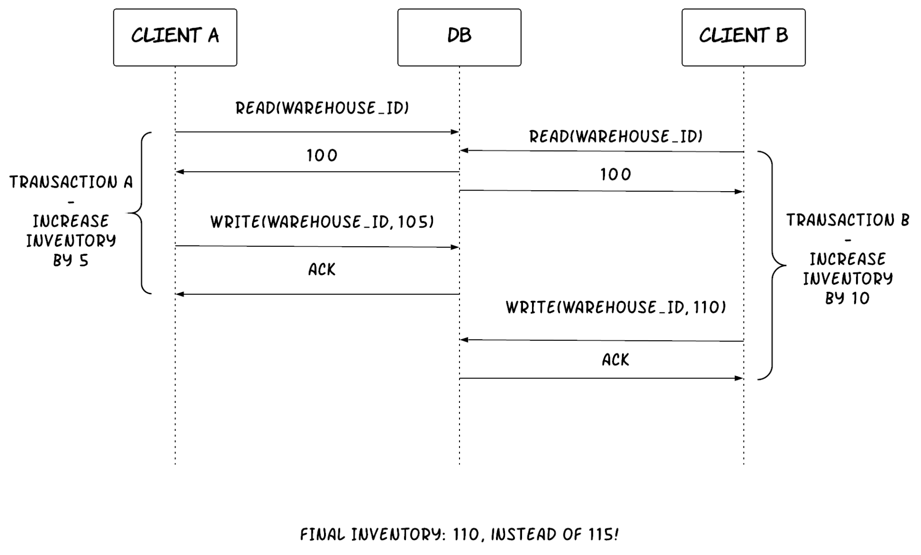

CLIENT A
DB
CLIENT B
READ(WAREHOUSE_ID)
100
READ(WAREHOUSE_ID)
100
TRANSACTION A - INCREASE INVENTORY BY 5
WRITE(WAREHOUSE_ID, 105)
ACK
TRANSACTION B - INCREASE INVENTORY BY 10
WRITE(WAREHOUSE_ID, 110)
ACK
FINAL INVENTORY: 110, INSTEAD OF 115!

Figure 2.9: Example of a lost update

A **read skew** occurs when there are integrity constraints between two data items that seem to be violated because a transaction can only see partial results of another transaction. For example, let's imagine an application that contains a table of persons, where each record represents a person and contains a list of all the friends of this person. The main integrity constraint is that friendships are mutual, so if person B is included in person A's list of friends, then A must also be included in B's list. Everytime someone (say P1) wants to unfriend someone else (say P2), a transaction is executed that removes P2 from P1's list and also removes P1 from P2's list at a single go. Now, let's also assume that some other part of the application allows people to view friends of multiple people at the same time. This is done by a transaction that reads the friends list of these people. If the second transaction reads the friends list of P1 before the first transaction has started, but it reads the friends list of P2 after the second transaction has committed, then it will notice an integrity violation. P2 will be in P1's list of friends, but P1 will not be in P2's list of friends. Note that this case is not a dirty read, since any values written by the first transaction are read only after it has been committed. See Figure 2.10 for a visualisation of this example. A strict requirement to prevent read skew is quite rare, as you might have guessed already. For example, a common application of this type might allow a user to view the profile of only one person at a time along with his or her friends, thus not having a requirement for the integrity constraint described above.

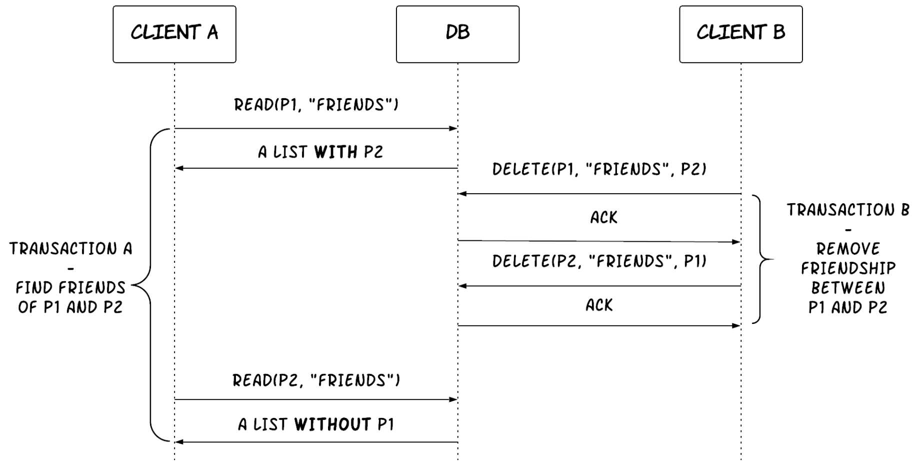

CLIENT A
DB
CLIENT B
READ(P1, "FRIENDS")
A LIST WITH P2
DELETE(P1, "FRIENDS", P2)
ACK
DELETE(P2, "FRIENDS", P1)
ACK
READ(P2, "FRIENDS")
A LIST WITHOUT P1
TRANSACTION A - FIND FRIENDS OF P1 AND P2
TRANSACTION B - REMOVE FRIENDSHIP BETWEEN P1 AND P2

Figure 2.10: Example of a read skew

A **write skew** occurs when two transactions read the same data, but then modify disjoint sets of data. For example, imagine of an application that maintains the on-call rota of doctors in a hospital. A table contains one record for every doctor with a field indicating whether they are oncall. The application allows a doctor to remove himself/herself from the on-call rota if another doctor is also registered. This is done via a transaction that reads the number of doctors that are on-call from this table and if the number is greater than one, then it updates the record corresponding to this doctor to not be on-call. Now, let's look at the problems that can arise from write skew phenomena. Let's say two doctors, Alice and Bob, are on-call currently and they both decide to see if they can remove themselves. Two transactions running concurrently might read the state of the database, seeing there are two doctors and removing the associated doctor from being on-call. In the end, the system ends with no doctors being on-call! See Figure 2.11 for a visualisation of this example.

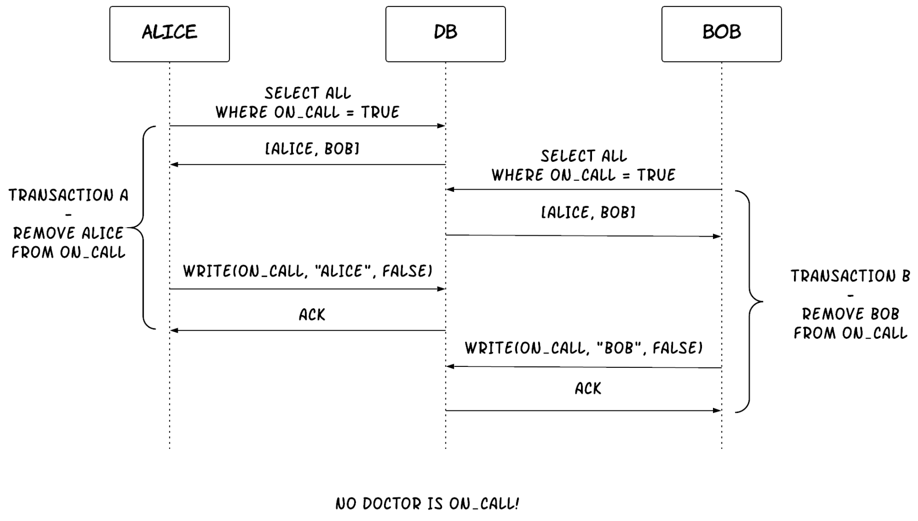
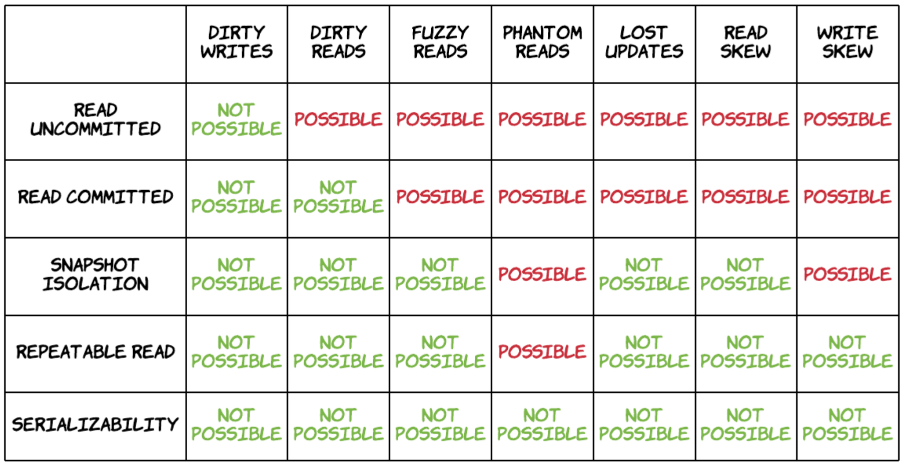

ALICE
DB
BOB
SELECT ALL WHERE ON_CALL = TRUE
[ALICE, BOB]
SELECT ALL WHERE ON_CALL = TRUE
TRANSACTION A - REMOVE ALICE FROM ON_CALL
[ALICE, BOB]
WRITE(ON_CALL, "ALICE", FALSE)
TRANSACTION B - REMOVE BOB FROM ON_CALL
ACK
WRITE(ON_CALL, "BOB", FALSE)
ACK
NO DOCTOR IS ON_CALL!

Figure 2.11: Example of a write skew

It must be obvious by now that there are so many different anomalies to consider. On top of that, different applications manipulate data in different ways, so one would have to analyse each case separately to see which of those anomalies could create problems.

Of course, there is one isolation level that prevents all of these anomalies, the **serializable** one[18][19]. Similar to the consistency models presented previously, this level provides a more formal specification of what is possible, e.g. which execution histories are possible. More specifically, it guarantees that the result of the execution of concurrent transactions is the same as that produced by some serial execution of the same transactions. This means that one can only analyse serial executions for defects. If all the possible serial executions are safe, then any concurrent execution by a system at the serializable level will also be safe.

However, serializability has performance costs, since it intentionally reduces concurrency to guarantee safety. The other less strict levels provide better performance via increased concurrency at the cost of decreased safety. These models allow some of the anomalies we described previously. Figure 2.12 contains a table with the most basic isolation levels along with the anomalies they prevent. As mentioned before, these isolation levels originated from the early relational database systems that were not distributed, but they are applicable in distributed datastores too, as shown later in the book.



	DIRTY WRITES	DIRTY READS	FUZZY READS	PHANTOM READS	LOST UPDATES	READ SKEW	WRITE SKEW
READ UNCOMMITTED	NOT POSSIBLE	POSSIBLE	POSSIBLE	POSSIBLE	POSSIBLE	POSSIBLE	POSSIBLE
READ COMMITTED	NOT POSSIBLE	NOT POSSIBLE	POSSIBLE	POSSIBLE	POSSIBLE	POSSIBLE	POSSIBLE
SNAPSHOT ISOLATION	NOT POSSIBLE	NOT POSSIBLE	NOT POSSIBLE	POSSIBLE	NOT POSSIBLE	NOT POSSIBLE	POSSIBLE
REPEATABLE READ	NOT POSSIBLE	NOT POSSIBLE	NOT POSSIBLE	POSSIBLE	NOT POSSIBLE	NOT POSSIBLE	NOT POSSIBLE
SERIALIZABILITY	NOT POSSIBLE	NOT POSSIBLE	NOT POSSIBLE	NOT POSSIBLE	NOT POSSIBLE	NOT POSSIBLE	NOT POSSIBLE

Figure 2.12: Isolation levels and prevented anomalies

Consistency and Isolation - Differences and Similarities

It is interesting to observe that isolation levels are not that different from consistency models. Both of them are essentially constructs that allow us to express what executions are possible or not possible. In both cases, some of the models are stricter allowing less executions, thus providing increased safety at the cost of reduced performance and availability. For instance, *linearizability* allows a subset of the executions *causal consistency* allows, while *serializability* also allows a subset of the executions *snapshot isolation* does. This strictness relationship can be expressed in a different way, saying that one model implies another model. For example, the fact that a system provides linearizability automatically implies that the same system also provides causal consistency. Note that there are some models that are not directly comparable, which means neither of them is stricter than the other.

At the same time, consistency models and isolation levels have some differences with regards to the characteristics of the allowed and disallowed behaviours. A main difference between the consistency models and the isolation levels presented so far is that the consistency models applied to **single-object** operations (e.g. read/write to a single register), while isolation levels applied to **multi-object** operations (e.g. read & write from/to multiple rows in a table within a transaction). Looking at the strictest models in these two groups, linearizability and serializability, there is another important difference. Linearizability provides **real-time guarantees**, while serializability does not. This means that linearizability guarantees that the effects of an operation took place at some point between the time the client invoked the operation and the result of the operation was returned to the client. Serializability only guarantees that the effects of multiple transactions will be the same as if they run in a serial order, but it does not provide any guarantee on whether that serial order would be compatible with real-time order.

Figure 2.13 contains the illustration of an example why real-time guarantees are actually important from an application perspective. Think of an automated teller machine that can support 2 transactions: `getBalance()` and `withdraw(amount)`. The first transaction performs a single operation to read the balance of an account. The second operation reads the balance of an account, reduces it by the specified amount and then returns to the client the specified amount in cash. Let's also assume this system is serializable. Now, let's examine the following scenario: a customer with an initial balance of `x` reads his/her balance and then decides to withdraw 20 dollars by executing a `withdraw(20)` transaction. After the transaction has been completed and the money is returned, the customer performs a `getBalance()` operation to check the new balance. However, the machine still returns `x` as the current balance, instead of `x-20`. Note that this execution is serializable and the end result is as if the machine executed first the `getBalance()` transactions and then the `withdraw(20)` transaction in a completely serial order. This example shows why in some cases serializability is not sufficient in itself.

In fact, there is another model that is a combination of linearizability and serializability, called **strict serializability**. This model guarantees that the result of multiple transactions is equivalent to the result of a serial execution of them that would also be compatible with the real-time ordering of these transactions. As a result, transactions appear to execute serially and the effects of each one of them takes place at some point between its invocation and its completion.

As illustated before, strict serializability is often a more useful guarantee

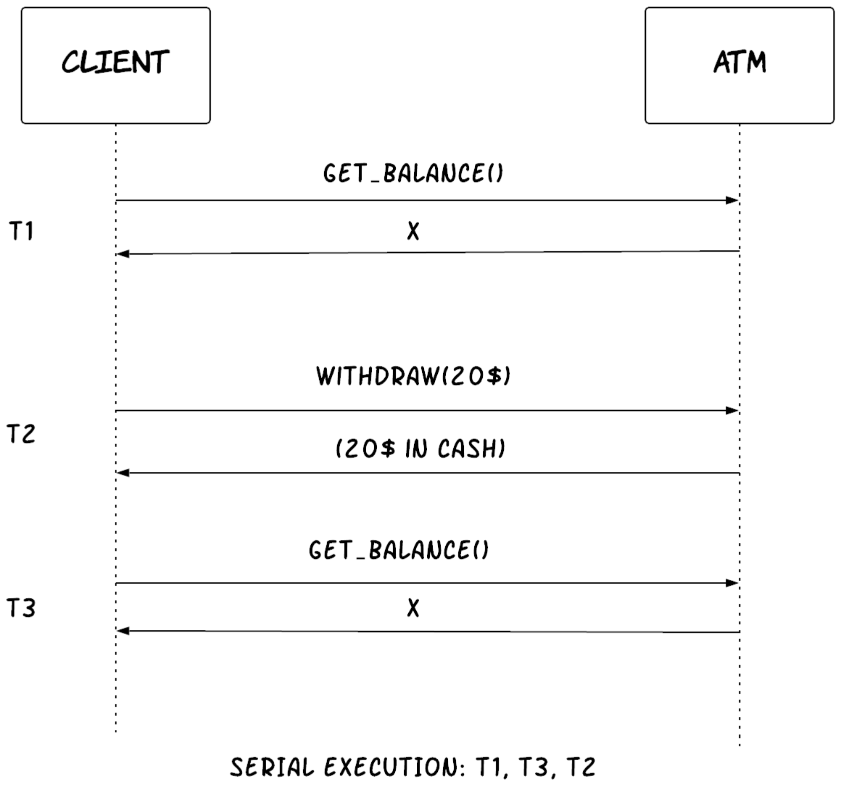


Figure 2.13: Why serializability is not enough sometimes

than plain serializability. However, in centralized systems providing strict serializability is simple and as efficient as only providing serializability guarantees. As a result, sometimes systems, such as relational databases, advertise serializability guarantees, while they actually provide strict serializability. This is not necessarily true in a distributed database, where providing strict serializability can be more costly, since additional coordination is required. Therefore, it is important to understand the difference between these two guarantees in order to determine which one is needed depending on the application domain.

All of the models presented so far - and many more that were not presented in this book for practical reasons - can be organised in a hierarchical tree according to their strictness and the guarantees they provide. This has actually been done in previous research [15]. Figure 2.14 contains such a tree containing only the models presented in this book.

Why all the formalities

The previous chapters spent significant amount of time going through many different formal models. But why do we need all these complicated, formal, academic constructs?

As explained before, these constructs help us define different types of properties in a more precise way. As a result, when designing a system it is easier to reason about what kind of properties the system needs to satisfy and which of these models are sufficient to provide the required guarantees. In many cases, applications are built on top of pre-existing datastores and they derive most of their properties from these datastores, since most of the data management is delegated to them. As a consequence, necessary research needs to be done to identify datastores that can provide the guarantees the application needs.

Unfortunately, the terminology presented here and the associated models are not used consistently across the industry making decision making and comparison of systems a lot harder. For example, there are datastores that do not state precisely what kind of consistency guarantees their system can provide or at least these statements are well hidden, while they should be highlighted as one of the most important things in their documentation. In some other cases, this kind of documentation exists, but the various levels presented before are misused leading to a lot of confusion. As mentioned

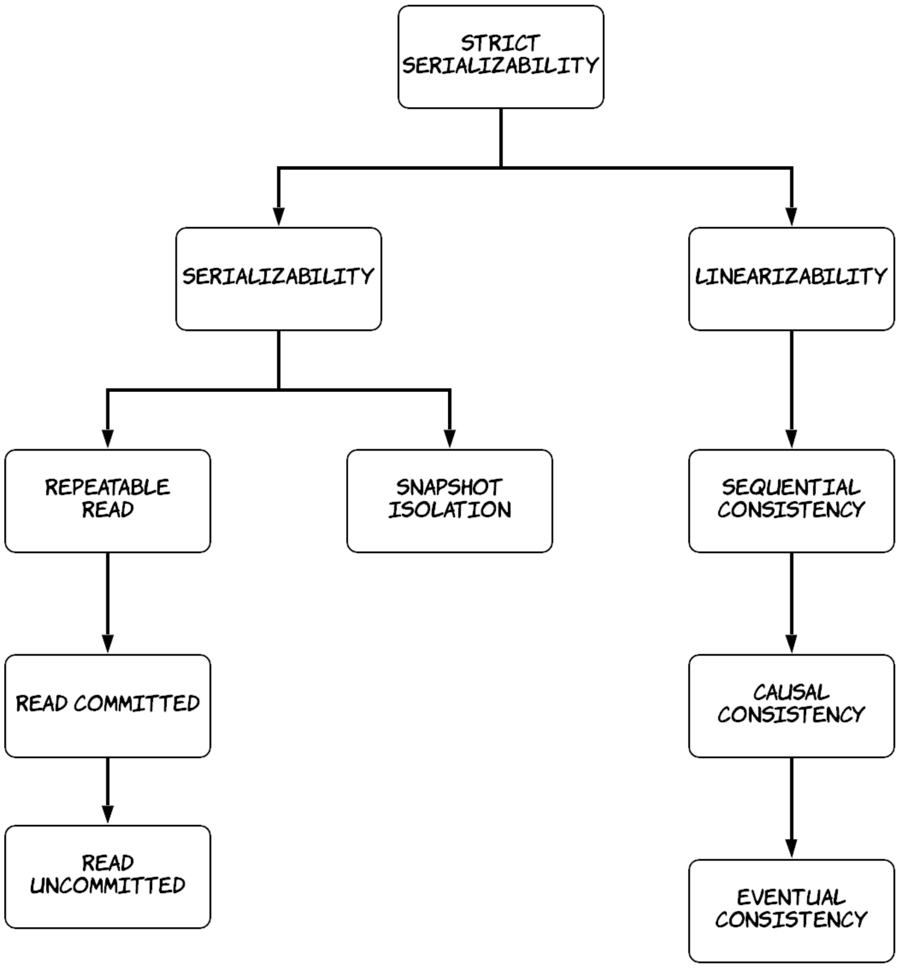


Figure 2.14: Hierarchy of consistency

before, one source of this confusion was the initial ANSI-SQL standard. For example, the `SERIALIZABLE` level provided by Oracle 11g, mySQL 5.7 and postgreSQL 9.0 was not truly serializable and was susceptible to some anomalies.

Understanding the models presented here is a good first step in thinking more carefully when designing systems to reduce risk for errors. You should be willing to search the documentation of systems you consider using to understand what kind of guarantees they provide. Ideally, you should also be able to read between the lines and identify mistakes or incorrect usages of terms. This will help you make more informed decisions. Hopefully, it will also help raise awareness across the industry and encourage vendors of distributed systems to specify the guarantees their system can provide.

Part II

Distributed Transactions & Consensus

Chapter 3

Distributed Transactions

One of the most common problems faced when moving from a centralised to a distributed system is performing some operations across multiple nodes in an atomic way, what is also known as a **distributed transaction**. In this chapter, we will explore all the complexities involved in performing a distributed transaction, examining several available solutions and the pitfalls of each one.

What is a distributed transaction

Before diving on the available solutions, let's first make a tour of transactions, their properties and what distinguishes a distributed transaction.

A transaction is a unit of work performed in a database system, representing a change, which can be potentially composed of multiple operations. Database transactions are an abstraction that has been invented in order to simplify engineers' work and relieve them from dealing with all the possible failures, introduced by the inherent unreliability of hardware.

As described previously, the major guarantees provided by database transactions are usually summed up in the acronym ACID, which stands for:

- Atomicity
- Consistency
- Isolation
- Durability

As we mentioned previously, each transaction TX can be composed of multiple operations (op_1, op_2, op_3, ...) and multiple transactions TX_1, TX_2 etc. can be executed simultaneously in a database. *Atomicity* is the property that guarantees that either all of the operations of a transaction will complete successfully or none of them will take effect. In other words, there are no situations where the transaction "partially fails", performing only some of the operations. *Consistency* ensures that a transaction will transition the database from a valid state to another valid state, maintaining all the invariants defined by the application. As an example, a financial application could define an invariant that states that the balance of every account should always be positive. The database then ensures that this invariant is maintained at all times, while executing transactions. *Isolation* guarantees that transactions can be executed concurrently without interfering with each other. *Durability* guarantees that once a transaction has been committed, it will remain committed even in the case of a failure system (i.e. power outage).

Each of those properties transfers a set of responsibilities from the application to the database, simplifying the development of applications and reducing the potential errors, because of software bugs or hardware failures. Atomicity implies that our application will not have to take care of all possible failures and have conditional logic in order to bring the database back to a consistent state in the case of a partial failure, rolling back operations that should not have taken effect. Consistency allows us to state the invariants of our application in a declarative way, removing redundant code from our application and allowing the database to perform these checks, when necessary. Isolation allows our applications to leverage concurrency, serving multiple requests by executing transactions in parallel, while being certain that the database will take care to prevent any bugs because of this concurrent execution. Last but not least, durability guarantees that when the database has declared a transaction committed, this will be a final declaration that can't be reverted, relieving again our application from complicated logic.

The aforementioned aspects of consistency and durability do not require special treatment in distributed systems and are relatively straightforward, so there will not be a separate analysis in this book. Consistency is maintained by many different mechanisms provided by databases, such as constraints, cascades, triggers etc. The application is responsible for defining any constraints through these mechanisms and the database is responsible for checking these conditions, while executing transactions, aborting any transactions that violate them. Durability is guaranteed by persisting transactions at non-volatile storage when they commit. In distributed systems, this might be a bit more nuanced, since the system should ensure that results of a transaction are stored in more than one node, so that the system can keep functioning if a single node fails. In fact, this would be reasonable, since availability is one of the main benefits of a distributed system, as we described in the beginning of this book. This is achieved via replication as described previously.

As we just explained, a database transaction is a quite powerful abstraction, which can simplify significantly how applications are built. Given the inherent complexity in distributed systems, one can easily deduce that transactional semantics can be even more useful in distributed systems. In this case, we are talking about a **distributed transaction**, which is a transaction that takes place in 2 or more different nodes. We could say that there are 2 slightly different variants of distributed transactions. The first variant is one, where the same piece of data needs to be updated in multiple replicas. This is the case where the whole database is essentially duplicated in multiple nodes and a transaction needs to update all of them in an atomic way. The second variant is one, where different pieces of data residing in different nodes need to be updated atomically. For instance, a financial application might be using a partitioned database for the accounts of customers, where the balance of user `A` resides in node `n1`, while the balance of user `B` resides in node `n2` and we want to transfer some money from user `A` to user `B`. This needs to be done in an atomic way, so that data are not lost (i.e. removed from user `A`, but not added in user `B`, because the transaction failed midway). The second variant is the most common use of distributed transactions, since the first variant is mostly tackled via single-master synchronous replication.

The aspects of atomicity and isolation are significantly more complex and require more things to be taken into consideration in the context of distributed transactions. For instance, partial failures make it much harder to guarantee atomicity, while the concurrency and the network asynchrony present in distributed systems make it quite challenging to preserve isolation between transactions running in different nodes. Furthermore, they can have far-reaching implications for the performance and the availability of a distributed system, as we will see later in this chapter.

For this reason, this book contains dedicated sections for these two aspects, analysing their characteristics and some techniques that can be used. Note that some of these techniques are used internally by database systems you might want to use from of your application to store and access data. This means that everything might be hidden from you behind a high-level interface that allows you to build an application without having to know how the database provides all these capabilities under the hood. It is still useful to have a good understanding of these techniques, because you can make better decisions about which database systems to use. There can also be cases where you might have to use some of these techniques directly at the application level to achieve properties that might not be provided by the database system.

Achieving Isolation

Previous sections have described some potential anomalies from concurrent transactions that are not properly isolated. Some examples were also provided that illustrated what the consequences can be in real life from these anomalies. As explained in that section, in order to be completely protected against any of these anomalies, the system should be *strictly serializable.*[1] This section will cover some algorithms that can be used to achieve strict serializability.

As explained in previous chapters, a system that provides serializability guarantees that the result of any allowed execution of transactions is the same as that produced by some serial execution[2] of the same transactions, hence the name. You might ask: what does *same* mean in the previous sentence? There are two major types of serializability that establish two different notions of similarity:

- **view serializability**: a schedule is *view* equivalent to a serial schedule with the same transactions, when all the operations from transactions in the two schedules read and write the same data values ("view" the same data).
- **conflict serializability**: a schedule is *conflict* equivalent to a serial schedule with the same transactions, when every pair of conflicting operations between transactions are ordered in the same way in both schedules.

[1]Note that this applies to transactions that are composed of multiple operations, as explained before. In cases where transactions are composed of a single operation, there is no potential for operations interleaving due to concurrency. In most of these cases, the system needs to satisfy linearizability or some of the weaker consistency models for single-object *operations.*

[2]In the context of isolation, an execution of multiple transactions that corresponds to an ordering of the associated operations is also called a **shedule**.

It turns out calculating whether a schedule is view serializable is a computationally very hard problem[3], so we will not analyse view serializability further. However, determining whether a schedule is conflict serializable is a much easier problem to solve, which is one of the reasons conflict serializability is widely used. In order to understand conflict serializability, we first have to define what it means for two operations to be conflicting.

Two operations are conflicting (or conflict) if:

- they belong to different transactions.
- they are on the same data item and at least one of them is a write operation, where a write operation inserts, modifies or deletes an object.

As a result, we can have three different forms of conflicts:

- a read-write conflict
- a write-read conflict
- a write-write conflict

A trivial way to check if a schedule is conflict serializable would be to calculate all possible serial schedules, identify conflicting operations in them and check if their order is the same as in the schedule under examination. As you might have thought already, this would be computationally heavy, since we would need to compute all the possible permutations of transactions. A more practical way of determining whether a schedule is conflict serializable is through a **precedence graph**.

A precedence graph is a directed graph, where nodes represent transactions in a schedule and edges represent conflicts between operations. The edges in the graph denote the order in which transactions must execute in the corresponding serial schedule. As a result, a schedule is conflict serializable if and only if its precedence graph of committed transactions is acyclic. Let's look at an example to get some intuition about this rule.

Figure 3.1 contains a schedule of three transactions T_1, T_2 and T_3, where $R(I_i)$ and $W(I_i)$ represent a read or write operation on item I_i respectively. As we can see in the diagram, the conflicting operations in this schedule form a precedence graph with a cycle. This means that this schedule is not conflict serializable. The cycle between T_1 and T_2 means that there must be a serial schedule where T_1 executes before T_2 and vice-versa, which

[3]More specifically, it is an NP-complete problem, which means the time required to solve the problem using any *currently known* algorithm increases rapidly as the size of the problem grows.

is impossible. Figure 3.2 contains a slightly different schedule of the same transactions that is now conflict serializable, since there are no cycles in the corresponding precedence graph. Looking at the precedence graph, we can see the edges only impose the constraints that T_1 must be before T_2 and T_3 in the corresponding serial schedule. This means that this schedule is conflict equivalent to both serial schedules T_1, T_2, T_3 and T_1, T_3, T_2.

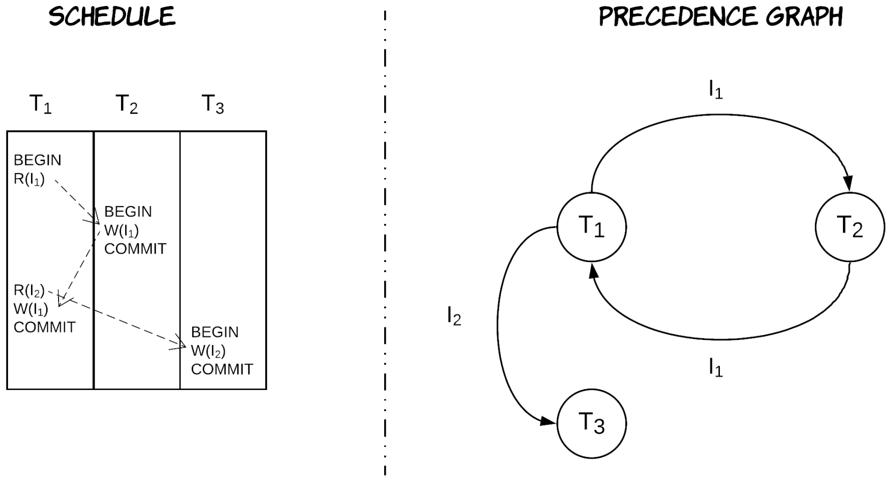


Figure 3.1: Precedence graph of a non-conflict serializable schedule

The method described above is one way to determine whether a schedule is serializable after the fact. However, what we really need is a way to generate a schedule that is serializable ahead of time. The notion of precedence graph is still very useful. All we need to do is ensure that as we execute operations in the schedule, no cycle is formed. This can be achieved in two basic ways. One way is to prevent transactions from making progress when there is a risk of introducing a conflict that can create a cycle. Another way is to let transactions execute all their operations and check if committing that transaction could introduce a cycle. In that case, the transaction can be aborted and restarted from scratch.

These two approaches lead to the two main mechanisms for concurrency control:

- **Pessimistic concurrency control**: this approach blocks a transaction if it's expected to cause violation of serializability and resumes it when it is safe to do so. This is usually achieved by having transactions

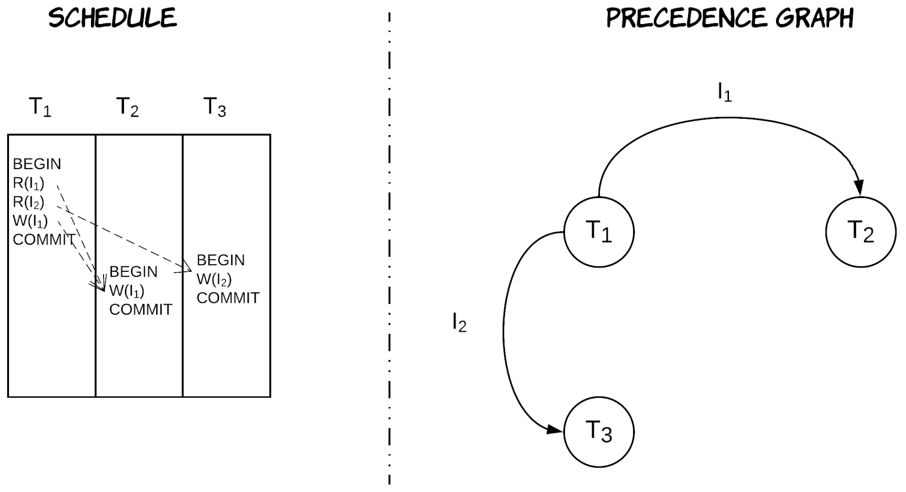


Figure 3.2: Precedence graph of a conflict serializable schedule

acquire locks on the data they process to prevent other transactions from processing the same data concurrently. The name *pessimistic* comes from the fact that this approach assumes that the majority of transactions are expected to conflict with each other, so appropriate measures are taken to prevent this from causing issues.

- **Optimistic concurrency control**: this approach delays the checking of whether a transaction complies with the rules until the end of the transaction. The transaction is aborted if a commit would violate any serializability rules and then it can be restarted and re-executed from the beginning. The name *optimistic* comes from the fact that this approach assumes that the majority of transactions are expected to not have conflicts, so measures are taken only in the rare case that they do.

The main trade-off between pessimistic and optimistic concurrency control is between the extra overhead from locking mechanisms and the wasted computation from aborted transactions. In general, optimistic methods are expected to perform well in cases where there are not many conflicts between transactions. This can be the case for workloads with many read-only transactions and only a few write transactions or in cases where most of the transactions touch different data. Pessimistic methods incur some overhead from the use of locks, but they can perform better in workloads that contain a lot of transactions that conflict, since they reduce the number of aborts & restarts, thus reducing wasted effort.

To understand better the mechanics and trade-offs of these two approaches, we will examine some algorithms from both of these categories in the following sections.

2-phase locking (2PL)

2-phase locking (2PL) is a pessimistic concurrency control protocol that makes use of locks to prevent concurrent transactions from interfering. These locks indicate that a record is being used by a transaction, so that other transactions can determine whether it is safe to use it or not.

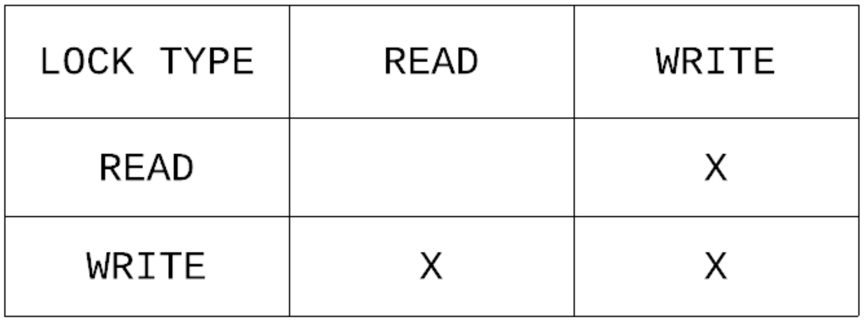
There are 2 basic types of locks used in this protocol:

- **Write (exclusive) locks**: these locks are acquired when a record is going to be written (inserted/updated/deleted).
- **Read (shared) locks**: these locks are acquired when a record is read.

The interaction between these 2 types of locks is the following:

- A read lock does not block a read from another transaction. This is the reason they are also called *shared*, because multiple read locks can be acquired at the same time.
- A read lock blocks a write from another transaction. The other transaction will have to wait until the read operation is completed and the read lock released, then it will have to acquire a write lock and perform the write operation.
- A write lock blocks both reads and writes from other transactions, which is also the reason it's also called *exclusive*. The other transactions will have to wait for the write operation to complete and the write lock to be released, then they will attempt to acquire the proper lock and proceed.

If a lock blocks another lock, they are called *incompatible*. Otherwise, they are called *compatible*. As a result, the relationships described above can be visualised in a *compatibility matrix*, as shown in Figure 3.3. The astute reader might notice a similarity between this matrix and the definition of conflicts in conflict serializability. This is not a coincidence, the two-phase locking protocol makes use of these locks to prevent cycles of these conflicts from forming, as described before. Each type of conflict is represented by an incompatible entry in this matrix.



LOCK TYPE	READ	WRITE
READ		X
WRITE	X	X

Figure 3.3: Compatibility of locks in 2PL

In this protocol, transactions acquire and release locks in 2 distinct phases:

- **Expanding phase**: in this phase, a transaction is allowed to only acquire locks, but not release any locks.
- **Shrinking phase**: in this phase, a transaction is allowed to only release locks, but not acquire any locks.

It's been implied so far that locks are held per-record. However, it's important to note that if the associated database supports operations based on predicates, there must also be a way to lock ranges of records (*predicate locking*), e.g. all the customers of age between 23 and 29. This is in order to prevent anomalies like phantom reads.

This protocol is proven to only allow serializable executions to happen [20]. A schedule generated by two-phase locking will be conflict equivalent to a serial schedule, where transactions are serialized in the order they completed their expanding phase. There are some slight variations of the protocol that can provide some additional properties, such as strict two-phase locking (S2PL) or strong strict two-phase locking (SS2PL).

The locking mechanism introduces the risk for deadlocks, where two transactions might be waiting on each other for the release of a lock thus never making progress. In general, there are two ways to deal with these deadlocks:

- prevention: prevent the deadlocks from being formed in the first place.

For example, this can be done if transactions know all the locks they need in advance and they acquire them in an ordered way. This is typically done by the application, since many databases support interactive transactions and are thus unaware of all the data a transaction will access.

- detection: detect deadlocks that occur and break them. For example, this can be achieved by keeping track of which transaction a transaction waits on, using this information to detect cycles that represent deadlocks and then forcing one of these transactions to abort. This is typically done by the database without the application having to do anything extra.

Optimistic concurrency control (OCC)

Optimistic concurrency control (OCC) is a concurrency control method that was first proposed in 1981[21], where transactions can access data items without acquiring locks on them. In this method, transactions execute in the following three phases:

- **begin**
- **read & modify**
- **validate & commit/rollback**

In the first phase, transactions are assigned a unique timestamp that marks the beginning of the transaction. During the second phase, transaction execute their read and write operations *tentatively*. This means that when an item is modified, a copy of the item is written to a temporary, local storage location. A read operation first checks for a copy of the item in this location and returns this one, if it exists. Otherwise, it performs a regular read operation from the database. The transaction enters the last phase, when all operations have executed. During this phase, the transaction checks whether there are other transactions that have modified the data this transaction has accessed and have started after this transaction's start time. If that's true, then the transaction is aborted and restarted from the beginning, acquiring a new timestamp. Otherwise, the transaction can be committed. The commit of a transaction is performed by copying all the values from write operations from the local storage to the common database storage that other transactions access. It's important to note that the validation checks and the associated commit operation need to be performed in a single atomic action as part of a critical section[4]. This requires essentially some form of locking mechanism, so there are various optimisations of this approach that attempt to reduce the duration of this phase in order to improve performance.

There are different ways to implement the validation logic:

- One way is via version checking, where every data item is marked with a version number. Every time a transaction accesses an item, it can keep track of the version number it had at that point. During the validation phase, the transaction can check the version number is the same, which would mean that no other transaction has accessed the item in the meanwhile.
- Another way is by using timestamps assigned to transactions, a technique also known as **timestamp ordering** since the timestamp indicates the order in which a transaction must occur, relative to the other transaction. In this approach, each transaction keeps track of the items that are accessed by read or write operations, known as the read set and the write set. During validation, a transaction performs the following inside a critical section. It records a fresh timestamp, called the finish timestamp, and iterates over all the transactions that have been assigned a timestamp between the transaction's start and finish timestamp. These are essentially all transactions that have started after the running transaction and have already committed. For each of those transactions, the running transaction checks if their write set intersect with its own read set. If that's true for any of these transactions, this means that the transaction essentially read a value *"from the future"*. As a result, the transaction is invalid and must be aborted and restarted from the beginning with a fresh timestamp. Otherwise, the transaction is committed and it's assigned the next timestamp.

If you are interested in more details or optimised versions of this protocol, reading the original paper would be a very good starting point[21].

[4]A critical section is some part of a program that can only be executed by only one process at a time, because it accesses shared resources for which concurrent access can lead to erroneous behaviour. In this case, this could happen if some other transaction commits in between the validation and commit of this transaction, which would make the validation results of the this transaction invalid.

Multi-version concurrency control (MVCC)

Multiversion Concurrency Control (MVCC) is a technique where multiple physical versions are maintained for a single logical data item. As a result, update operations do not overwrite existing records, but they just write new version of these records. Read operations can then select a specific version of a record, possibly an older one. This is in contrast with the previous techniques, where updates are performed in-place and there is a single record for each data item that can be accessed by read operations. The original protocol was first proposed in a dissertation in 1978[22], but many different variations of the original idea have been proposed since then[23][24]. As the name implies, this technique is focused on the multi-version aspect of storage, so it can be used theoretically with both optimistic and pessimistic schemes. However, most variations use an optimistic concurrency control method in order to leverage the multiple versions of an item from transactions that are running concurrently.

In practice, MVCC is commonly used to implement the snapshot isolation level. As explained before, *Snapshot Isolation (SI)* is an isolation level that essentially guarantees that all reads made in a transaction will see a consistent snapshot of the database from the point it started and the transaction will commit successfully if no other transaction has updated the same data since that snapshot. As a result, it is practically easier to achieve snapshot isolation using an MVCC technique.

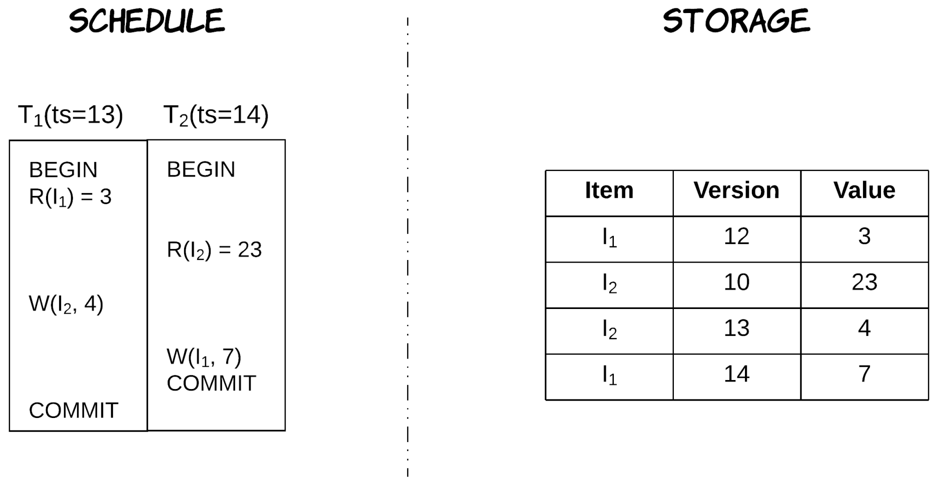
This works in the following way:

- Each transaction is assigned a unique timestamp at the beginning.
- Every entry for a data item contains a version that corresponds to the timestamp of the transaction that created this new version.
- Every transaction records the following pieces of information during its beginning: the transaction with the highest timestamp that has committed so far (let's call it T_s) and the number of active transactions that have started but haven't been commited yet.
- When performing a read operation for an item, a transaction returns the entry with the latest version that is earlier than T_s and does not belong to one of the transactions that were active at the beginning of this transaction. This prevents dirty reads as only committed values from other transactions can be returned. There is an exception to this rule: if the transaction has already updated this item, then this value is returned instead. Fuzzy reads are also prevented, since all the reads

will return values from the same snapshot and will ignore values from transactions that committed after this transaction started.

- When performing a write operation for an item, a transaction checks whether there is an entry for the same item that satisfies one of the following criteria: its version is higher than this transaction's timestamp or its version is lower than this transaction's timestamp, but this version belongs to one of the transactions that were active at the beginning of this transaction. In any of these cases, the transaction is aborted and can be restarted from scratch with a larger timestamp. In the first case, if the transaction commmitted, then we would have an entry with version T_j committed before an entry with version T_i even though $T_i < T_j$, which is wrong. In the second case, the transaction is aborted to prevent a lost update anomaly.

As explained already, this prevents a lot of the anomalies, but it is still not serializable and some anomalies would still be possible. An example of such an anomaly that would not be prevented is write skew. Figure 3.4 contains an example that illustrates how this can happen. In the schedule shown by this example, none of the transactions sees the versions written by the other transaction. However, this would not be possible in a serial execution.



Item	Version	Value
I_1	12	3
I_2	10	23
I_2	13	4
I_1	14	7

Figure 3.4: Write skew in MVCC

Research on this field has resulted in an improved algorithm, called **Seri-**

alizable Snapshot Isolation (SSI), which can provide full serializability [25] and has been integrated in commercial, widely used databases [26]. This algorithm is still optimistic and just adds some extensions on top of what has been described above.

The mechanics of the solution are based on a key principle of previous research that showed that all the non-serializable executions under snapshot isolation share a common characteristic. This states that in the precedence graph of any non-serializable execution, there are two *rw-dependency* edges which form consecutive edges in a cycle and they involve two transactions that have been active concurrently, as shown in Figure 3.5. A *rw-dependency* is a data dependency between transactions T_1 and T_2 where T_1 reads a version of an item x and T_2 produces a version of item x that is later in the version order than the version read by T_1. This was the case in the example presented previously in Figure 3.4 too.

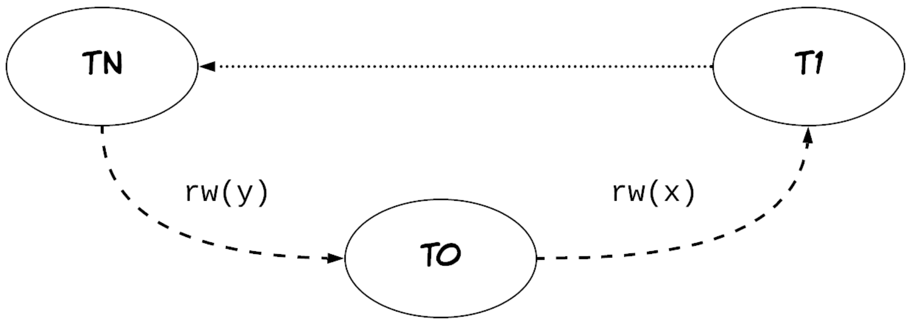


Figure 3.5: Dangerous structure in serialization graph for SSI

This approach detects these cases and breaks the cycle when they are about to happen and it prevents them from being formed, by aborting one of the involved transactions. In order to do so, it keeps track of the incoming and outgoing rw-dependency edges of each transaction. If there is a transaction that has both incoming and outgoing edges, the algorithm aborts one of the transactions and retries it later.[5] So, it is sufficient to maintain two boolean flags per transaction `T.inConflict` and `T.outConflict` denoting whether there is an incoming and outgoing rw-dependency edge. These flags can be

[5]Note this can lead to aborts that are false positives, since the algorithm does not check whether there is a cycle. This is done intentionally to avoid the computational costs associated with tracking cycles.

maintained in the following way:

- When a transaction `T` is performing a read, it is able to detect whether there is a version of the same item that was written after the transaction started, e.g. by another transaction `U`. This would imply a rw-dependency edge, so the algorithm can update `T.outConflict` and `U.inConflict` to true.
- However, this will not detect cases where the write happens after the read. The algorithm uses a different mechanism to detect these cases too. Every transaction creates a read lock, called *SIREAD lock*, when performing a read. As a result, when a transaction performs a write it can read the existing SIREAD locks and detect concurrent transactions that have previously read the same item, thus updating accordingly the same boolean flags. Note that these are a *softer* form of locks, since they do not block other transactions from operating, but they exist mainly to signal data dependencies between them. This means the algorithm preserves its optimistic nature.

Figure 3.6 shows how this approach would prevent the write skew anomaly in our previous example. When transaction T_2 executes the write operation, it checks for existing SIREAD locks on item I_1. T_1 holds such a lock, so transaction T_2 updates its `inConflict` flag to true. Given both the `inConflict` and the `outConflict` flags for T_2 are true at this moment, this transaction is aborted.

For brevity, this explanation omitted some details of SSI. If you are interested, have a look at the related papers[25][26].

Achieving atomicity

As explained before, the second challenging aspect of transactions, and especially distributed transactions, is atomicity. One of the benefits of grouping operations inside a transaction is the guarantee that either all of them will be performed or none of them will be performed. As a result, the application developer does not need to think about scenarios of partial failures, where the transaction has failed midway after some of the operations have been performed. Similar to the isolation guarantees, this makes it easier to develop applications and delegates some of the complexity around handling these situations to the persistence layer, e.g. to the datastore used by the application that provides atomicity guarantees.

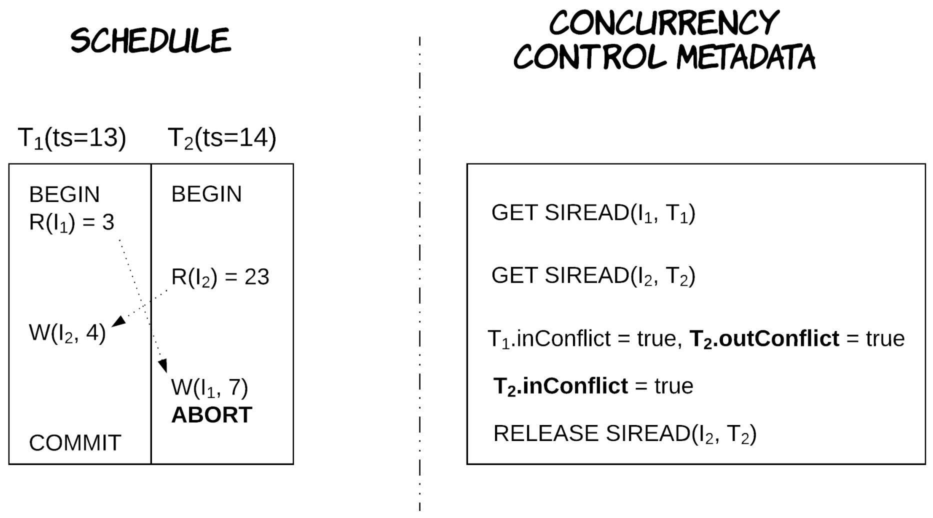


Figure 3.6: How write skew is prevented in SSI

Guaranteeing atomicity is hard in general, not only in distributed systems. The reason is components can fail unexpectedly regardless of whether they are software or hardware components. Even the simple action of writing some bytes to a file requires extra work to ensure it will be performed in an atomic way and the file will not end up in a corrupted state if the disk fails while executing part of the operation [27]. One common way of achieving atomicity in this case is by using a technique called *journalling* or *write-ahead logging*, where metadata about the operation are first written to a separate file along with markers that denote whether an operation has been completed or not. Based on this data, the system is capable of identifying which operations were in-progress when a failure happened and drive them to completion either by undoing their effects and aborting them or by completing the remaining part and committing them. This approach is used extensively in file systems and databases.

The issue of atomicity in a distributed system becomes even more complicated, because components (nodes in this context) are separated by the network that is slow and unreliable, as explained already. Furthermore, we do not only need to make sure that an operation is performed atomically in a node, but in most cases we need to ensure that an operation is performed atomically across multiple nodes. This means that the operation needs to take effect either at all the nodes or at none of them. This problem is also known as **atomic commit**[6].

In the next sections, we will be looking at how atomicity can be achieved in distributed settings. Algorithms are discussed in chronological order, so that the reader can understand what are the pitfalls of each algorithm and how they were addressed by subsequent ones.

2-phase commit (2PC)

As we've described many times so far, in a distributed system with an unreliable network, just sending a message to the involved nodes would not be enough for executing a distributed transaction. The node initiating the transaction would be unable to know whether the other nodes committed successfully or aborted because of some failure in order to take a final decision about the result of the transaction. Thinking about that, the simplest idea is to add another round of messages, checking what was the result on each node.

This is essentially the **2-phase commit** protocol (*2PC*) [28][29], which consists of 2 phases, as described previously, hence the name. The protocol contains 2 different roles: the *coordinator* and the *participants*. Their names reflect their actual responsibilities in the protocol, with the first being responsible for coordinating the different phases of the protocol and the second corresponding to all the nodes that participate in the transaction. Note that one of the participants could also play the role of the coordinator. As described, the protocol contains 2 phases:

- The first phase is called the *voting phase*, where the coordinator sends the transaction to all the participants. The participants execute the transaction up to the point where they are supposed to commit.[7] Then, they respond to the coordinator with a vote that shows if the

[6]See: https://en.wikipedia.org/wiki/Atomic_commit

[7]In some cases, the operations of each transaction are executed separately and before the voting phase, which starts after all the operations of a transaction have been executed. Furthermore, agreement protocols like this usually involve some locking protocol as well, so that other concurrent transactions cannot make participants that have already voted change their mind on whether they can commit or not. For example, the 2-phase commit protocol can be combined with the 2-phase locking protocol.

transaction's operations were executed successfully (Yes vote) or there was some error that means the transaction can't be committed (No vote).

- The second phase is called the *commit phase*, where the coordinator gathers all the votes from the participants. If all the votes were Yes, then the coordinator messages the participants again with an instruction to commit the transaction. Otherwise, if at least one vote is No, the coordinator instructs the participants to abort the transaction. Finally, the participants reply with an acknowledgement, closing this phase.

The fact that a unanimous positive vote is needed for commit means that the transaction will either commit to all the participants or it will be aborted to all of them (atomicity property). The coordinator and the participants make use of a write-ahead-log, where they persist their decisions during the various steps, so that they can recover in case of a failure. The coordinator also uses a timeout, when waiting for the responses from the first phase. If the timeout expires, the coordinator interprets this timeout as a No vote, considering the node has failed. On the other hand, the participants do not apply any timeouts, while waiting for coordinator's messages since that could lead to participants reaching different conclusions due to timing issues. Figure 3.7 shows what this flow looks like.

Since the happy path is straightforward, let's examine how the protocol handles various kinds of failures:

- Failure of a participant in the first phase: As we described before, if a participant fails in the voting phase before replying to the coordinator, the coordinator will time out waiting and assume a No vote on behalf of this participant. This means that the protocol will end up aborting the transaction.
- Failure of a participant in the second phase: In this scenario, a participant votes in the first phase, but then fails, before receiving the message from the coordinator and completing the transaction (either by commit or abort). In this case, the protocol will conclude without this node. If this node recovers later on, it will identify that pending transaction and will communicate with the coordinator to find out what was the result and conclude it in the same way. So, if the result of the transaction was successful, any crashed participant will eventually find out upon recovery and commit it, the protocol does not allow aborting it unilaterally.[8] Thus, atomicity is maintained.

[8]An astute reader will observe that there is a chance that the participants might fail

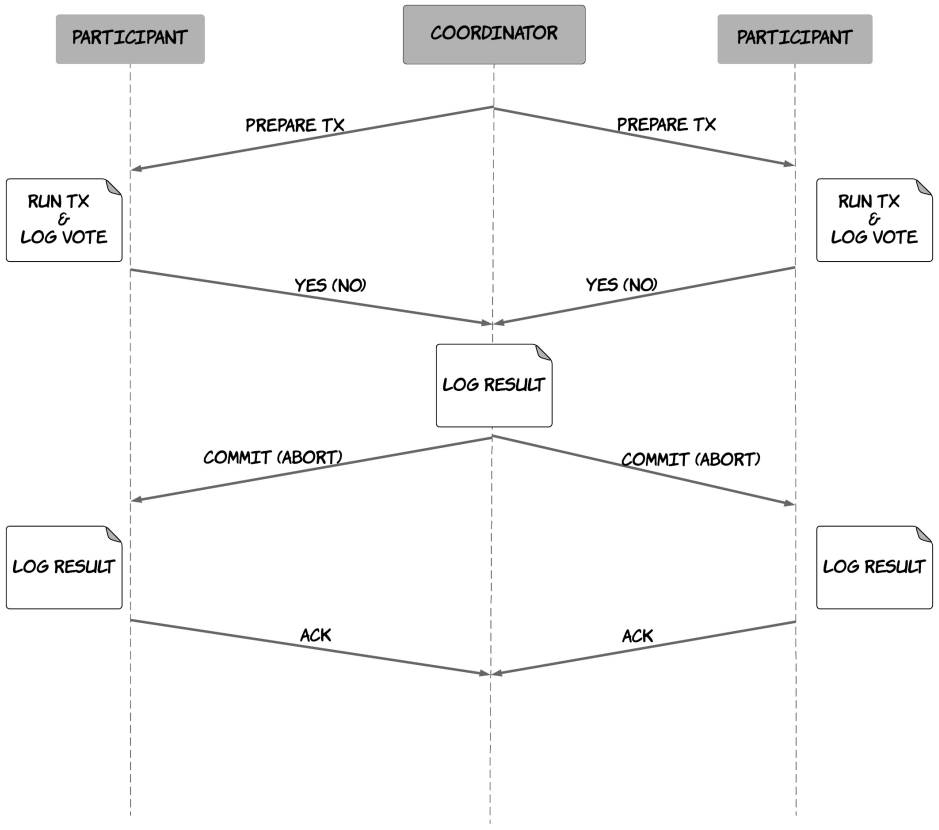


Figure 3.7: 2-phase commit flow

- Network failures in the same steps of the protocol have similar results to the ones described previously, since timeouts will make them equivalent to node failures.

Even though 2-phase commit can handle gracefully all the aforementioned failures, there's a single point of failure, the coordinator. Because of the blocking nature of the protocol, failures of the coordinator at specific stages of the protocol can bring the whole system to a halt. More specifically, if a coordinator fails after sending a prepare message to the participants, then the participants will block waiting for the coordinator to recover in order to find out what was the outcome of the transaction, so that they commit or abort it respectively. This means that failures of the coordinator can decrease availability of the system significantly. Moreover, if the data from the coordinator's disk cannot be recovered (e.g. due to disk corruption), then the result of pending transactions cannot be discovered and manual intervention might be needed to unblock the protocol.

Despite this, the 2-phase commit has been widely used and a specification for it has also been released, called the *eXtended Architecture* (*XA*)[9]. In this specification, each of the participant nodes are referred to as *resources* and they must implement the interface of a *resource manager*. The specification also defines the concept of a *transaction manager*, which plays the role of the coordinator starting, coordinating and ending transactions.

To conclude, the 2PC protocol satisfies the **safety** property that all participants will always arrive at the same decision (atomicity), but it does not satisfy the **liveness** property that it will always make progress.

3-phase commit (3PC)

As we described previously, the main bottleneck of the 2-phase commit protocol was failures of the coordinator leading the system to a blocked state. Ideally, we would like the participants to be able to take the lead in some

at the point they try to commit the transaction essentially breaking their promise, e.g. because they are out of disk space. Indeed, this is true, so participants have to make the minimum work possible as part of the commit phase to avoid this. For example, the participants can write all the necessary data on disk during the first phase, so that they can signal a transaction is committed by doing minimal work during the second phase (e.g. flipping a bit).

[9]See: https://en.wikipedia.org/wiki/X/Open_XA

way and continue the execution of the protocol in this case, but this is not so easy. The main underlying reason is the fact that in the 2nd phase, the participants are not aware of the state of the other participants (only the coordinator is), so taking the lead without waiting for the coordinator can result in breaking the atomicity property. For instance, imagine the following scenario: in the second phase of the protocol, the coordinator manages to send a commit (or abort, respectively) message to one of the participants, but then fails and this participant also fails. If one of the other participants takes the lead, it will only be able to query the live participants, so it will be unable to make the right decision without waiting for the failed participant (or the coordinator) to recover.

This problem could be tackled by splitting the first round into 2 sub-rounds, where the coordinator first communicates the votes result to the nodes, waiting for an acknowledgement and then proceeds with the commit/abort message. In this case, the participants would know what was the result from the votes and could complete the protocol in case of a coordinator failure independently. This is essentially the **3-phase commit protocol** (*3PC*) [30][31]. Wikipedia contains a nice, detailed description of the various stages of the protocol and a nice visual demonstration[10], so we won't repeat it here, but feel free to refer to this resource for additional study on the protocol. The main benefit of this protocol is that the coordinator stops being a single point of failure. In case of a coordinator failure, the participants are now able to take over and complete the protocol. A participant taking over can commit the transaction if it has received a prepare-to-commit, knowing that all the participants have voted yes. If it has not received a prepare-to-commit, it can abort the transaction, knowing that no participant has committed, without all the participants having received a prepare-to-commit message first.

As a result, it seems that the 3PC protocol increases availability, preventing the coordinator from being a single point of failure. However, this comes at the cost of correctness, since the protocol is vulnerable to some kind of failures, such as network partitions. An example of such a failure case is shown in Figure 3.8. In this example, a network partition occurs at a point, where the coordinator has managed to send a prepare-to-commit message only to some participants, while the coordinator fails right after this point, so the participants time out and have to complete the protocol on their own. In this case, the one side of the partition has participants that have received a

[10]See: https://en.wikipedia.org/wiki/Three-phase_commit_protocol

prepare-to-commit and continue with committing the transaction. However, the other side of the partition has not received a prepare-to-commit message and thus unilaterally abort the transaction. This can seem like a failure case that is very unlikely to happen. However, the consequences are disastrous if it does happen, since after the network partition is fixed, the system is now at an inconsistent state, where the atomicity property of the transaction has been violated.

To conclude, the 3PC protocol satisfies the **liveness** property that it will always make progress at the cost of violating the **safety** property of atomicity.

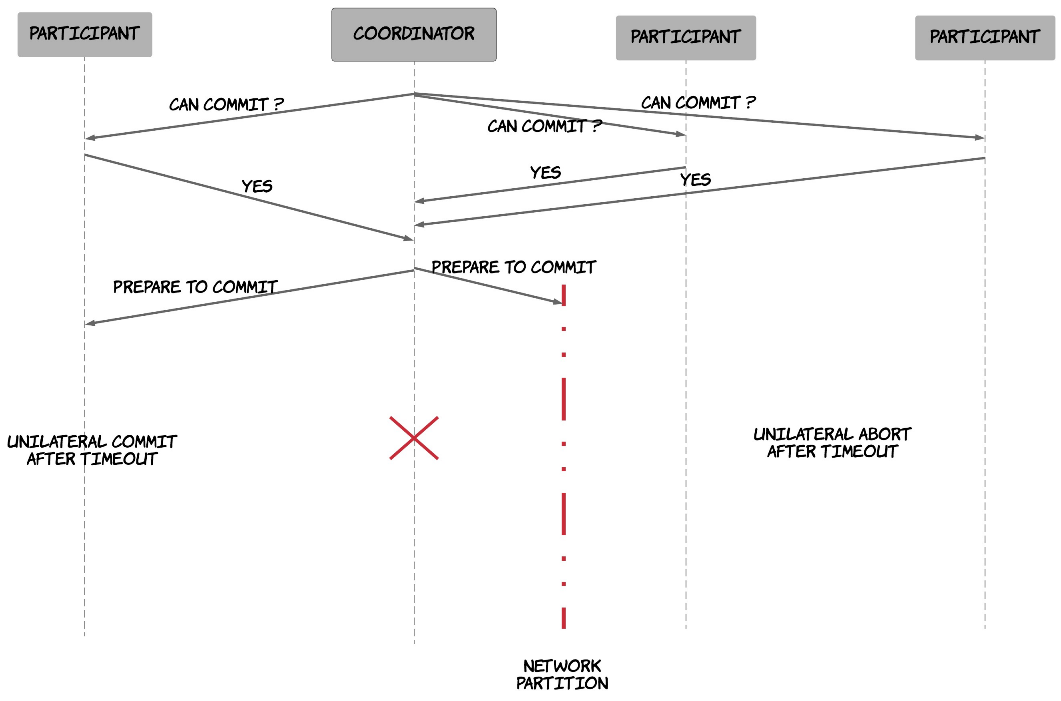


Figure 3.8: Network partition issues in 3PC

A quorum-based commit protocol

As we observed in the previous section, the main issue with the 3PC protocol occurs in the end of the second phase, where a potential network partition can bring the system to an inconsistent state. This can happen due to the fact that participants attempt to unblock the protocol, by taking the lead without having a picture of the overall system, resulting in a split-brain situation[11]. Ideally, we would like to be able to cope with this network partition, but without compromising on the safety of the protocol. This can be done using a concept we have already introduced in the book, a *quorum*.

This approach is followed by the **quorum-based commit protocol** [32]. This protocol is significantly more complex, when compared to the other two protocols we described previously, so you should study the original paper carefully, if you want to examine all the possible edge cases. However, we will attempt to give a high-level overview of the protocol in this section.

As we mentioned before, this protocol leverages the concept of a quorum to ensure that different sides of a partition do not arrive to conflicting results. The protocol establishes the concept of a commit quorum ($\mathrm{V_C}$) and an abort quorum ($\mathrm{V_A}$). A node can proceed with committing only if a commit quorum has been formed, while a node can proceed with aborting only if an abort quorum has been formed. The values of the abort and commit quorums have to be selected so that the property $\mathrm{V_A} + \mathrm{V_C} > \mathrm{V}$ holds, where V is the total number of participants of the transaction. Based on the fact that a node can be in only one of the two quorums, it's impossible for both quorums to be formed in two different sides of the partition, leading in conflicting results.

The protocol is composed of 3 different sub-protocols, used in different cases:

- the *commit protocol*, which is used when a new transaction starts
- the *termination protocol*, which is used when there is a network partition
- the *merge protocol*, which is used when the system recovers from a network partition

The commit protocol is very similar to the 3PC protocol. The only difference is that the coordinator is waiting for $\mathrm{V_C}$ number of acknowledgements in the end of the third phase to proceed with committing the transaction. If there is a network partition at this stage, then the coordinator can be rendered unable to complete the transaction. In this case, the participants

[11]See: https://en.wikipedia.org/wiki/Split-brain_(computing)

on each side of a partition will investigate whether they are able to complete the transaction, using the termination protocol. Initially, a (surrogate) coordinator will be selected amongst them via leader election. Note that which leader election algorithm is used is irrelevant and even if multiple leaders are elected, this does not violate the correctness of the protocol. The elected coordinator queries the nodes of the partition for their status. If there is at least one participant that has committed (or aborted), the coordinator commits (or aborts) the transaction, maintaining the atomicity property. If there is at least one participant at the prepare-to-commit state and at least V_C participants waiting for the votes result, the coordinator sends prepare-to-commit to the participants and continues to the next step. Alternatively, if there's no participant at the prepare-to-commit state and at least V_A participants waiting for the results vote, the coordinator sends a prepare-to-abort message. Note that this message does not exist in the commit protocol, but only in the termination one. The last phase of the termination protocol waits for acknowledgements and attempts to complete the transaction in a similar fashion to the commit protocol. The merge protocol is simple, including a leader election amongst the leaders of the 2 partitions that are merged and then execution of the termination protocol we described.

Let's examine what would happen in the network partition example from the previous section (Figure 3.8). In this case, we had 3 participants ($V = 3$) and we will assume that the protocol would use quorums of size $V_A = 2$ and $V_C = 2$. As a result, during the network partition, the participant on the left side of the partition would be unable to form a commit quorum. On the other hand, the participants on the right side of the partition would be able to form an abort quorum and they would proceed with aborting the transaction, assuming no more partitions happen. Later on, when the network partition recovers, the merge protocol would execute, ensuring that the participant from the left side of the partition would also abort the transaction, since the new coordinator would identify at least one node that has aborted the transaction. Figure 3.9 contains a visualisation of this execution. An interesting property of the protocol is that one can tune the values of the quorums V_A, V_C, thus effectively adjusting the protocol's tendency to complete a transaction via commit or abort in the presence of a partition.

To conclude, the quorum-based commit protocol satisfies the **safety** property that all participants will always arrive at the same decision (atomicity). It does not satisfy the **liveness** property that it will always make progress, since

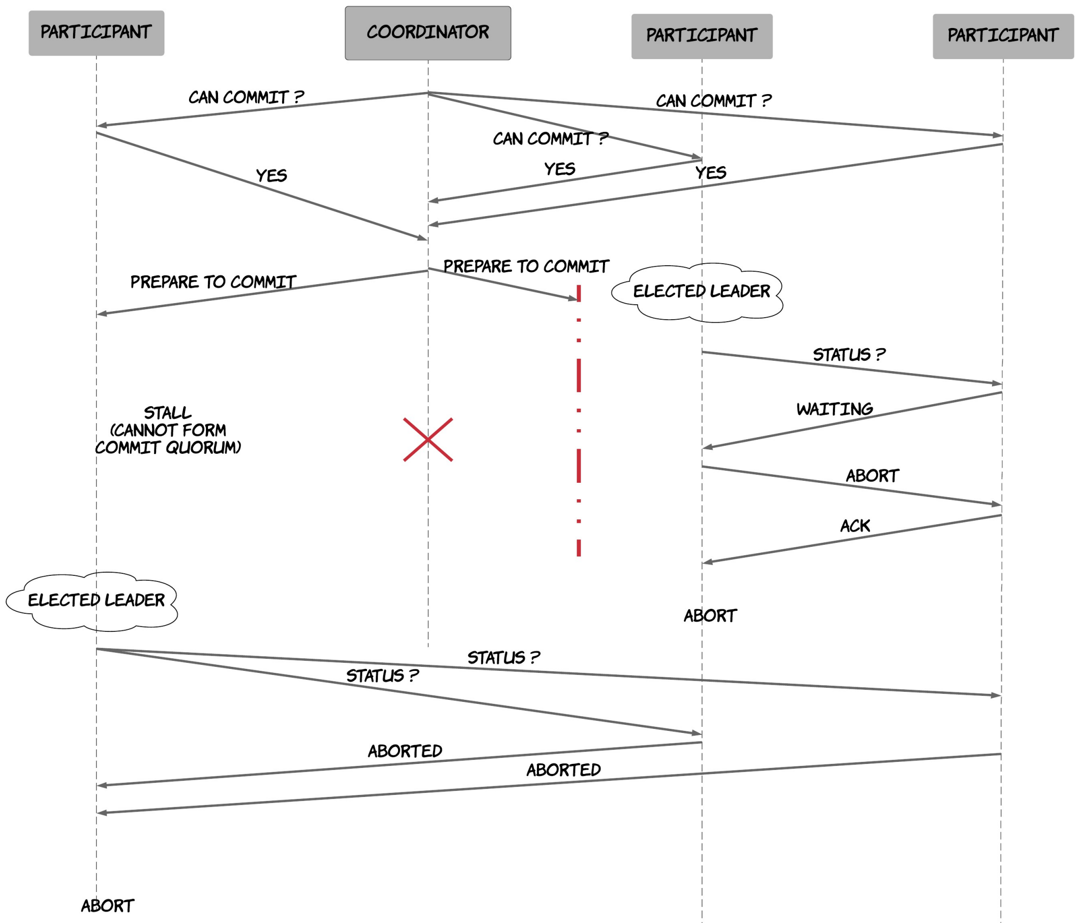


Figure 3.9: Network partition in quorum-based commit

there are always degenerate, extreme failure cases (e.g. multiple, continuous and small partitions). However, it's much more resilient, when compared to 2PC and other protocols and can make progress in the most common types of failures.

How it all fits together

As described already many times, transactions need to provide some guarantees if applications are to get some benefit out of using them. Distributed transactions need to provide similar guarantees. Some basic guarantees commonly used are contained in the ACID acronym that has been analysed previously. As explained before, consistency and durability do not require very different treatment in a distributed setting when compared to a centralised, single-node system. For durability, it's enough for the data to be stored in non-volatile storage before being acknowledged to the client. In a distributed system, it might be better to do this in more than one replicas before acknowledging, so that the system can survive failures of a single node. To achieve consistency, the system can just introduce some additional read & write operations in the transaction's context that exist to guarantee application consistency is preserved. These operations might be automatically generated, such as referential integrity contraints from foreign keys or cascades, or they might be defined by the application e.g. via triggers.

The guarantees of atomicity and isolation are more challenging to preserve and some of the algorithms that can be used were analysed previously. The book examined some algorithms that can help preserve isolation across transactions and some algorithms that can be used to preserve atomicity in a distributed system. These algorithms must be combined in order to guarantee all of these properties. Some combinations of these algorithms might be easier to implement in practice because of their common characteristics. For example, two-phase locking has very similar characteristics with two-phase commit and it's easier to understand how they can be combined. Spanner [5] is an example of a system that uses a combination of these two techniques to achieve atomicity and isolation, as explained later in the book.

Looking at the previous algorithms presented, it is not very hard to realise that some of them introduce either brittleness (e.g. two-phase commit) or a lot of additional complexity to a system (e.g. quorum-based commit). The algorithms presented for isolation can be used both in centralised and distributed systems. However, their use in a distributed system has several additional implications. For example, two-phase locking requires use of distributed locks, which is something that is not trivial to implement in a distributed system, as explained later in the book. Optimistic techniques, such as snapshot isolation, will require a lot of data transfer between different nodes in a distributed system, which will have adverse effects in terms of performance [33]. As a consequence, using transactions in a distributed system comes at a higher cost when compared to a centralised system and systems that do not have a strong need for them can be designed in such a way that makes it possible to operate safely without them. That is also one of the reasons why many distributed databases either do not provide full support for ACID transactions or force the user to explicitly opt in to use them.

Long-lived transactions & Sagas

As explained previously, achieving complete isolation between transactions is relatively expensive. The system either has to maintain locks for each transaction potentially blocking other concurrent transactions from making progress or it might have to abort some transactions to maintain safety which leads to some wasted effort. Furthermore, the longer the duration of a transaction is the bigger the impact of these mechanisms is expected to be on the overall throughput. There is also a positive feedback cycle in that using these mechanisms can cause transactions to take longer, which can increase the impact of these mechanisms.

In fact, there is a specific class of transactions, called **long-lived transactions** (*LLT*). These are transactions that are by their nature transactions that have a longer duration in the order of hours or even days, instead of milliseconds. This can happen because this transaction is processing a large amount of data, requires human input to proceed or needs to communicate with 3rd party systems that are slow. Examples of LLTs are:

- batch jobs that calculate reports over big datasets
- claims at an insurance company, containing various stages that require human input
- an online order of a product that spans several days from order to delivery

As a result, running these transactions using the common concurrent mechanisms would degrade performance significantly, since they would need to hold resources for long periods of time, while not operating on them. What's more, sometimes these transactions do not really require full isolation between each other, but they still need to be atomic, so that consistency is maintained under partial failures. Thus, researchers came up with a new concept, **the saga**[34]. The saga is essentially a sequence of transactions T_1, T_2, ..., T_N that can be interleaved with other transactions. However, it's guaranteed that either all of the transactions will succeed or none of them will, maintaining the atomicity guarantee. Each transaction T_i is associated with a so-called *compensating transaction* C_i, which is executed in case a rollback is needed.

The concept of saga transactions can be really useful in distributed systems. As demonstrated in the previous sections, distributed transactions are generally hard and can only be achieved, by making compromises on performance and availability. There are cases, where a saga transaction can be used instead of a distributed transaction, satisfying all of our business requirements while also keeping our systems loosely coupled and achieving good availability and performance.

As an example, let's imagine we are building an e-commerce application, where every order of a customer is composed of several discrete steps, such as credit card authorization, checking warehouse inventory, item shipping, invoice creation & delivery etc. One approach could be to perform a distributed transaction across all these systems for every order. However, in this case, failure of a single component (i.e. the payment system) could potentially bring the whole system to a halt, as we explained previously. An alternative, leveraging the saga pattern, would be to model the order operation as a saga operation, consisting of all these sub-transactions, where each of them is associated with a compensating transaction. For example, debiting a customer's bank account could have a compensating transaction that would give a refund. Then, we can build the order operation as a sequential execution of these transactions, as shown in Figure 3.10 . In case any of these transactions fails, then we rollback the transactions that have been executed, running their corresponding compensating transactions.

There might still be cases where some form of isolation is needed. In the example above, orders from different customers about the same product might share some data, which can lead to interference between each other. For instance, think about the scenario of 2 concurrent orders A and B, where A has reserved the last item from the warehouse. As a result of this, order

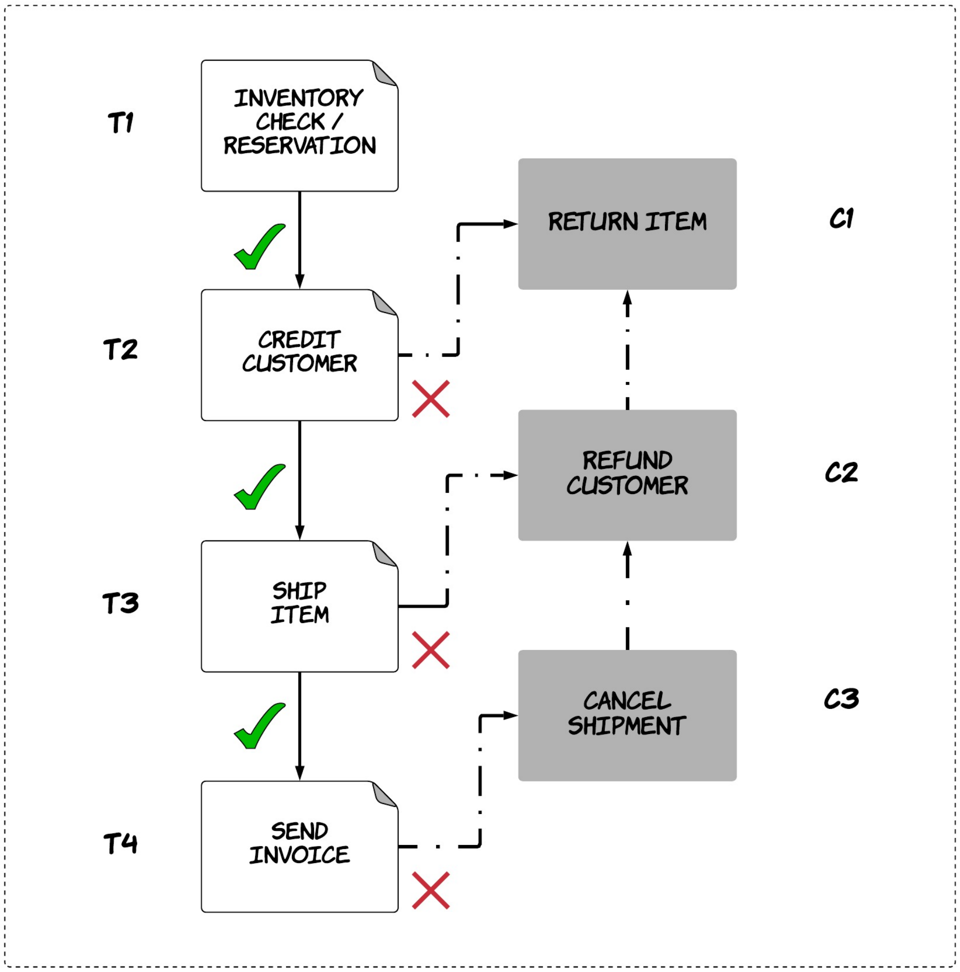


Figure 3.10: Example of a saga transaction

B fails at the first step and it's rejected because of zero inventory. Later on, order A also fails at the second step because the customer's card does not have enough money and the associated compensating transaction is run returning the reserved item to the warehouse. This would mean that an order would have been rejected while it could have been processed normally. Of course, this violation of isolation does not have severe consequences, but in some cases the consequences might be more serious, e.g. leading to customers being charged without receiving a product.

In order to prevent these scenarios, some form of isolation can be introduced at the application layer. This topic has been studied by previous research that proposed some concrete techniques [35], referred to as *countermeasures* to isolation anomalies. Some of these techniques are:

- the use of a *semantic lock*, which essentially signals that some data items are currently in process and should be treated differently or not accessed at all. The final transaction of a saga takes care of releasing this lock and resetting the data to their normal state.
- the use of *commutative updates* that have the same effect regardless of their order of execution. This can help mitigate cases that are otherwise susceptible to lost update phenomena.
- re-ordering the structure of the saga, so that a transaction called as a *pivot* transaction delineates a boundary between transactions that can fail and those that can't. In this way, transactions that can't fail - but could lead to serious problems if being rolled-back due to failures of other transactions - can be moved after the pivot transaction. An example of this is a transaction that increases the balance of an account. This transaction could have serious consequences if another concurrent saga reads this increase in the balance, but then the previous transaction is rolled back. Moving this transaction after the pivot transaction means that it will never be rolled back, since all the transactions after the pivot transaction can only succeeed.

These techniques can be applied selectively in cases where they are needed. However, they introduce significant complexity and move some of the burden back to the application developer that has to think again about all the possible failures and design accordingly. These trade-offs need to be taken into consideration when choosing between using saga transactions or leveraging transaction capabilities of the underlying datastore.

Chapter 4

Consensus

Amongst all the problems encountered so far in the book, there is a common trait that characterizes most (if not all) of them. It's the fact that the various nodes of a distributed systems try to reach an agreement on a specific thing. In the case of a distributed transaction, it's whether a transaction has been committed or not. In case of a message delivery, it's whether a message has been delivered or not. In fact, this underlying property is common in many more problems in the distributed systems space. As a result, researchers have formally defined this problem and researched possible solutions, since these can then be used as a building block for more complicated problems. This is known as the **consensus** problem and this chapter of the book is devoted to it.

Defining consensus

First of all, we need to formally define the problem of consensus. We assume we have a distributed system, consisting of k nodes (n_1, n_2, ..., n_k), where each one can propose a different value v_i. Consensus is the problem of making all these nodes agree on a single value v. The following properties must also be satisfied:

- **Termination**: Every non-faulty node must eventually decide.
- **Agreement**: The final decision of every non-faulty node must be identical.

- **Validity**: The value that is agreed must have been proposed by one of the nodes.

Some use-cases of consensus

As explained before, the problem of consensus lies beneath a lot of other common problems in the distributed systems space. We will now visit some of them and discuss how they relate to the consensus problem.

A very common problem is **leader election**, where the nodes that are part of a distributed system need to elect one node amongst them to act as their leader, coordinating the operation of the whole system. An example of this is the single-master replication scheme that was presented previously in the book. This scheme is based on the fact that one node, designated as primary, will be responsible for performing operations that update data and the other nodes, designated as secondaries, will be following up with the same operations. However, in order to do that the system first needs to select the primary node, which is a process called leader election. Since, all the nodes are practically agreeing on a single value, the identity of the leader, this problem can easily be modelled as a consensus problem.

One more common problem is **distributed locking**. Most distributed systems receive multiple concurrent requests and need to perform some concurrency control to prevent data inconsistencies, because of interference between these requests. One of these concurrency control methods is locking, but using locking in the context of a distributed system comes with a lot of edge-cases, adding a lot of risk. Of course, distributed locking can also be modelled as a consensus problem, where the nodes of the system agree on a single value, which is the node that holds the lock.

Another commonly cited problem is **atomic broadcast**, which is concerned with allowing a set of nodes to concurrently broadcast messages while ensuring that all destinations consistently deliver them in the exact same sequence despite the possible presence of faulty nodes. This problem is also equivalent to consensus, as also demonstrated in previous research [6][36].

The reason we described these problems and demonstrated how they can be modelled as a consensus problem is so that you can appreciate the value of this abstraction and understand that solving the consensus problem can provide solutions to many more problems.

FLP impossibility

Researchers have found a lot of different solutions to this problem, but they have also found important constraints that impose some limitations on the possible solutions. We should note that it's extremely useful to know the limits of the available solutions to a problem and the research community has benefited massively from this. As a result, this chapter will unfold in a counter-intuitive way, first explaining these limitations and then discussing one of the solutions to the problem. In this way, we hope the reader will be able to gain a better understanding of the problem and will be better equipped to reason about the solution presented later on.

As explained previously in the book, there are several different system models with the *asynchronous* being the one that is really close to real-life distributed systems. So, it's been proved that in asynchronous systems, where there can be at least one faulty node, any possible consensus algorithm will be unable to terminate, under some scenarios. In other words, there can be no consensus algorithm that will always satisfy all the aforementioned properties. This is referred to as the *FLP impossibility* after the last initials of the authors of the associated paper[37]. The proof in the paper is quite complicated, but it's essentially based on the following 2 parts. First, the fact that it's always possible that the initial state of the system is one, where nodes can reach different decisions depending on the ordering of messages (the so-called *bivalent configuration*), as long as there can be at least one faulty node. Second, from such a state it's always possible to end up in another bivalent state, just by introducing delays in some messages.

As a result, **it's impossible to develop a consensus algorithm that will always be able to terminate successfully in asynchronous systems, where at least one failure is possible**. What we can do instead is develop algorithms that minimize this possibility of arriving at ambivalent situations.

The Paxos algorithm

Some of the algorithms we have already studied in previous chapters could arguably be applied as solutions to the consensus problem. For instance, the 2-phase commit protocol could be used, where the coordinator would drive the voting process. However, as we have already seen, such a protocol would have very limited fault-tolerance, since the failure of a single node (the coordinator) could bring the whole system to a halt. The obvious next step is to allow multiple nodes to inherit the role of the coordinator in these failure cases. This would then mean that there might be multiple masters that might produce conflicting results. We have already demonstrated this phenomenon in the chapter about multi-master replication and when explaining the 3-phase commit.

One of the first algorithms that could solve the consensus problem safely under these failures is called the **Paxos algorithm**. More specifically, this algorithm guarantees that the system will come to an agreement on a single value, tolerating the failure of any number of nodes (potentially all of them), as long as more than half the nodes are working properly at any time, which is a significant improvement. Funnily enough, this algorithm was invented by Leslie Lamport during his attempt to prove this is actually impossible! He decided to explain the algorithm in terms of a parliamentary procedure used in an ancient, fictional Greek island, called Paxos. Despite being elegant and highly entertaining, this first paper[38] was not well received by the academic community, who found it extremely complicated and could not discern its applicability in the field of distributed systems. A few years later and after several successful attempts to use the algorithm in real-life systems, Leslie decided to publish a second paper[39], explaining the algorithm in simpler terms and demonstrating how it can be used to build an actual, highly-available distributed system. A historical residue of all this is the fact that the Paxos algorithm is regarded as a rather complicated algorithm until today. Hopefully, this section will help dispel this myth.

The Paxos algorithm defines 3 different roles: the *proposers*, the *acceptors* and the *learners*. Every node in the system can potentially play multiple roles. A proposer is responsible for proposing values (potentially received from clients' requests) to the acceptors and trying to persuade them to accept their value in order to arrive at a common decision. An acceptor is responsible for receiving these proposals and replying with their decision on whether this value can be chosen or not. Last but not least, the learners are responsible for learning the outcome of the consensus, storing it (in a replicated way) and potentially acting on it, by either notifying clients about the result or performing actions. Figure 4.1 contains a visual overview of these roles and how they interact with the clients.

The algorithm is split into 2 phases, each of which contains two parts:

- **Phase 1 (a)**: A proposer selects a number `n` and sends a `prepare` request with this number (`prepare(n)`) to at least a majority of the

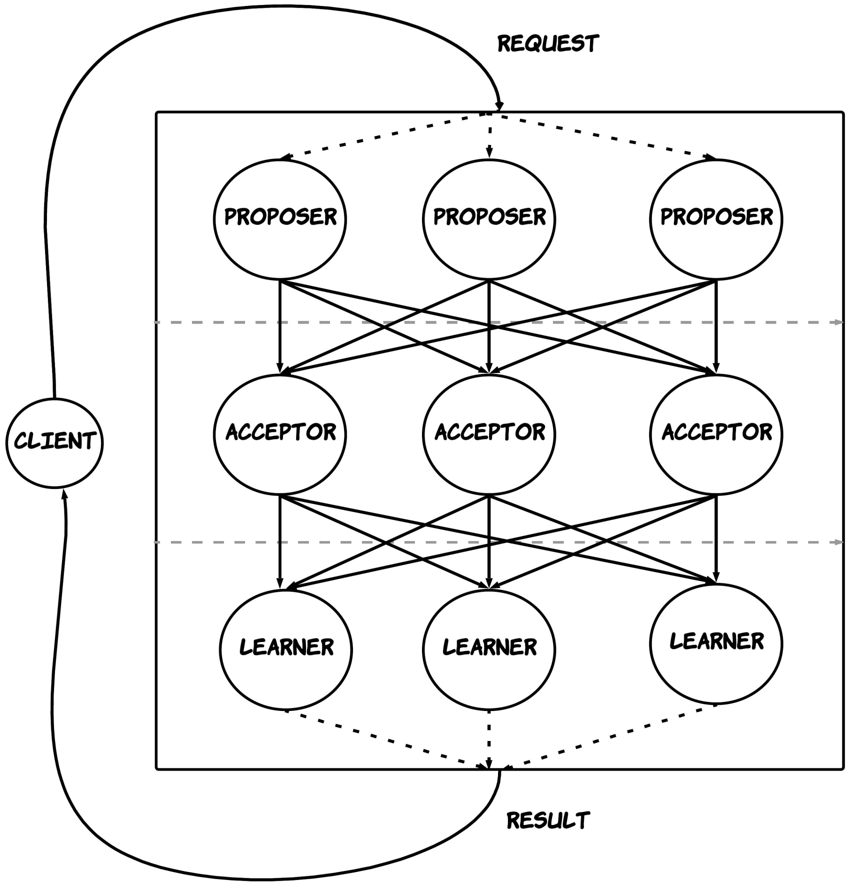


Figure 4.1: An overview of the paxos protocol

acceptors.

- **Phase 1 (b)**: When receiving a `prepare` request, an acceptor has the following options:
  - If it has not already responded to another `prepare` request of a higher number `n`, it responds to the request with a `promise` not to accept any more proposals that are numbered less than `n`. It also returns the highest-numbered proposal it has accepted, if any (note: the definition of a proposal follows).
  - Otherwise, if it has already accepted a `prepare` request with a higher number, it rejects this `prepare` request, ideally giving a hint to the proposer about the number of that other `prepare` request it has already responded to.
- **Phase 2 (a)**: If the proposer receives a response to its `prepare(n)` requests from a majority of acceptors, then it sends an `accept(n, v)` request to these acceptors for a proposal numbered `n` with a value `v`. The value is selected according to the following logic:
  - If any of the acceptors had already accepted another proposal and included that in its response, then the proposer uses the value of the highest-numbered proposal among these responses. Essentially, this means that the proposer attempts to bring the latest proposal to conclusion.
  - Otherwise, if none of the acceptors had accepted any other proposal, then the proposer is free to select any desired value. This value is usually selected based on the clients' requests.
- **Phase 2 (b)**: If the acceptor receives an `accept(n, v)` request for a proposal numbered `n`, it accepts the proposal, unless it has already responded to a `prepare(k)` request of a higher number (`k > n`).

Furthermore, as the acceptors accept proposals, they also announce their acceptance to the learners. When a learner receives an acceptance from a majority of acceptors, it knows that a value has been chosen. This is the most basic version of the Paxos protocol. As we mentioned previously, nodes can play multiple roles for practical reasons and this is usually the case in real-life systems. As an example, one can observe that the proposers can play the role of learners as well, since they will be receiving some of these accept responses anyway, thus minimising traffic and improving the performance of the system.

During Phase 1 (a) of the protocol, the proposers have to select a proposal number `n`. These numbers must be unique in order for the protocol to maintain its correctness properties. This is so that acceptors are always able to compare two `prepare` messages. This can be achieved in several ways, but the easiest one is to compose these numbers out of 2 parts, the one being an integer and the second one being a unique identifier of the proposer (i.e. the IP address of the node). In this way, proposers can draw numbers from the same set. As we have insinuated in the beginning of this section, multiple proposers can initiate concurrent `prepare` requests. The proposer that receives a response to its `prepare` request from a majority of acceptors is essentially elected as the current (but temporary) leader. As a result, it can proceed with making a proposal request. The value of this proposal will be the chosen one, unless a majority of acceptors have failed (and did not reply to the proposal) or another leader stepped up becoming the temporary leader in the meanwhile (in which case, the acceptors will reject this proposal).

The basic ingredient of the Paxos protocol is a concept we have already seen, namely the *quorum*. More specifically, the Paxos protocol makes use of majority quorums. A majority quorum is one that consists of more than half of the nodes of the system, i.e. at least `k+1` nodes in a system of `2k` nodes. The fact that proposers require a majority quorum to reply to their `prepare` requests to proceed with a proposal guarantees that there can't be 2 different proposers that complete both phases of the protocol concurrently. As a result, only a single value can be chosen, satisfying the *agreement* property of consensus.

Intricacies of Paxos

As we mentioned previously, many people consider the Paxos protocol to be difficult to understand. One of the reasons for this is the inherent complexity of the consensus problem, which in turn originates from the increased concurrency and large state space of distributed systems. This section will cover some edge cases and how Paxos handles them. Of course, we will not be able to cover all the possible cases, since that would be a much bigger undertaking. But, we hope the examples presented in this section will help you understand the basic parts of the protocol and give you a starting point for exploring any other cases you might think of. For all of these examples, we will assume that the nodes play all the roles of the protocol, thus being proposers, acceptors and learners at the same time, in order to simplify our explanations. Keep in mind that this a realistic assumption, since many implementations of the Paxos protocol follow this approach.

The beginning of this chapter outlined how Paxos can be used to solve the leader election problem. Nonetheless, Paxos itself needs to elect a leader in order to reach consensus, which seems like a catch-22[1]. The Paxos protocol resolves this paradox, by allowing multiple leaders to be elected, thus not needing to reach consensus for the leader itself. It still has to guarantee that there will be a single decision, even though multiple nodes might be proposing different values. Let's examine how Paxos achieves that and what are some of the consequences. When a proposer receives a response to a `prepare` message from a majority of nodes, it considers itself the *(temporary)* leader and proceeds with a proposal. If no other proposer has attempted to become the leader in the meanwhile, its proposal will be accepted. However, if another proposer managed to become a leader, the `accept` requests of the initial node will be rejected. This prevents multiple values to be chosen by the proposals of both nodes. This can result in a situation, where proposers are continuously duelling each other, thus not making any progress, as you can see in Figure 4.2. There are many ways to avoid getting into this infinite loop. The most basic one is forcing the proposers to use random delays or exponential back-off every time they get their `accept` messages rejected and have to send a new `prepare` request. In this way, they give more time to the node that is currently leading to complete the protocol, by making a successful proposal, instead of competing.

Another interesting aspect of the Paxos protocol is how it handles partial failures gracefully, maintaining safety at all times. In this context, by partial failures we refer to cases, where a node sends a message to multiple nodes (i.e. accept messages as part of Phase 2.a) and only some of them are delivered either due to node failures or network issues. As an example, let's examine an extreme case, where multiple proposers attempt to propose different values, but only one of their `accept` messages gets delivered to the acceptors of the majority quorum. Figure 4.3 provides a visualisation of the execution of the protocol to aid comprehension. Every row represents a different round of the protocol, while the dashed box shows which nodes were included in the majority quorum of Phase 1. The text inside every node displays any proposal that has already been accepted in the form `(n, v)`, where `n` is the proposal number and `v` is the value of the proposal. The bold text represents the values that have been accepted in that round. As you can see, every proposer manages to deliver an `accept` message to only one acceptor

[1]See: https://en.wikipedia.org/wiki/Catch-22_(logic)

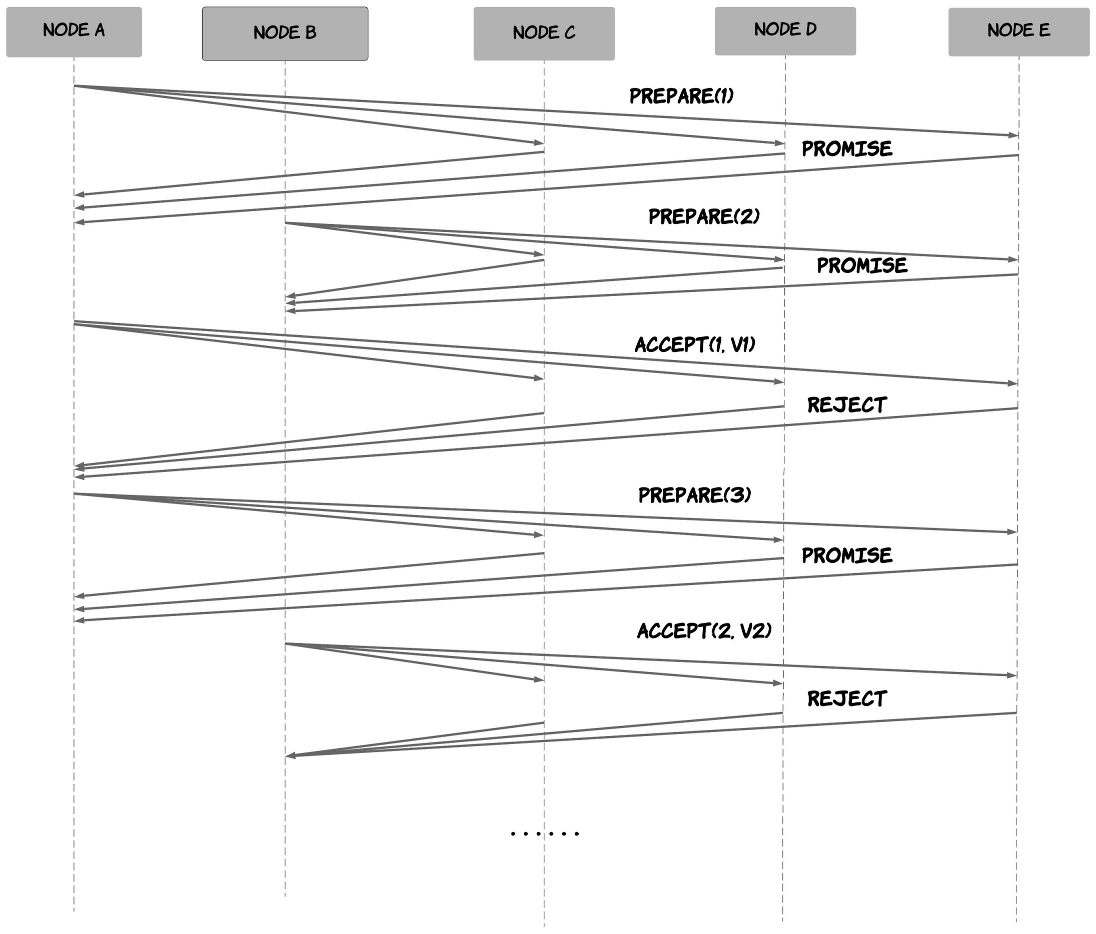


Figure 4.2: The situation of duelling proposers

at every round. For the first 3 rounds, none of the nodes in the majority quorum have accepted any value, so proposers are free to propose their own value. In rounds 4 and 5, proposers have to propose the value of the highest-numbered proposal that has been accepted by the acceptors included in Phase's 1 majority quorum. This is A for round 4 and B for round 5. As it's demonstrated for round 6, at this point the behaviour depends partially on the quorum that will be used. For example, if the next proposer selects the yellow quorum, value C is going to be proposed, while value B will be proposed if the green quorum is used instead. However, there is one important thing to note: as soon as the system recovers from failures and a proposer manages to get a proposal accepted by a majority quorum, then this value is chosen and it cannot be changed. The reason is that any subsequent proposer will need to get a majority quorum for Phase 1 of the protocol. This majority will have to contain at least 1 node from the majority that has accepted the aforementioned proposal, which will thus transfer the accepted proposal to the prospective leader. Furthermore, it's guaranteed this will be the highest-numbered proposal, which means any subsequent proposer can only propagate the chosen value to the acceptors that might not have it yet.

Paxos in real-life

As we have seen so far, the Paxos protocol is a well specified protocol. However, there are some small details and optimisations that the original paper could not cover. Some of these topics have been covered in subsequent papers [40]. This section will cover some of these topics as briefly as possible.

The basic Paxos protocol describes how a distributed system of multiple nodes can decide on a single value. However, just choosing a single value would have limited practical applications on its own. In order to be able to build more useful systems, we need to be able to continuously select values. This can be achieved by running multiple *instances* of Paxos, where an instance is an execution of the protocol that leads to a decision on a single value. These instances can run independently and in parallel, but they also have to be numbered. Depending on the functionality needed, there can be several rules applied, such as not returning the result of an instance to the client, unless all the previous instances have completed as well. We will elaborate more on this topic in the next section.

Another common need is the ability to query the current state of the system.

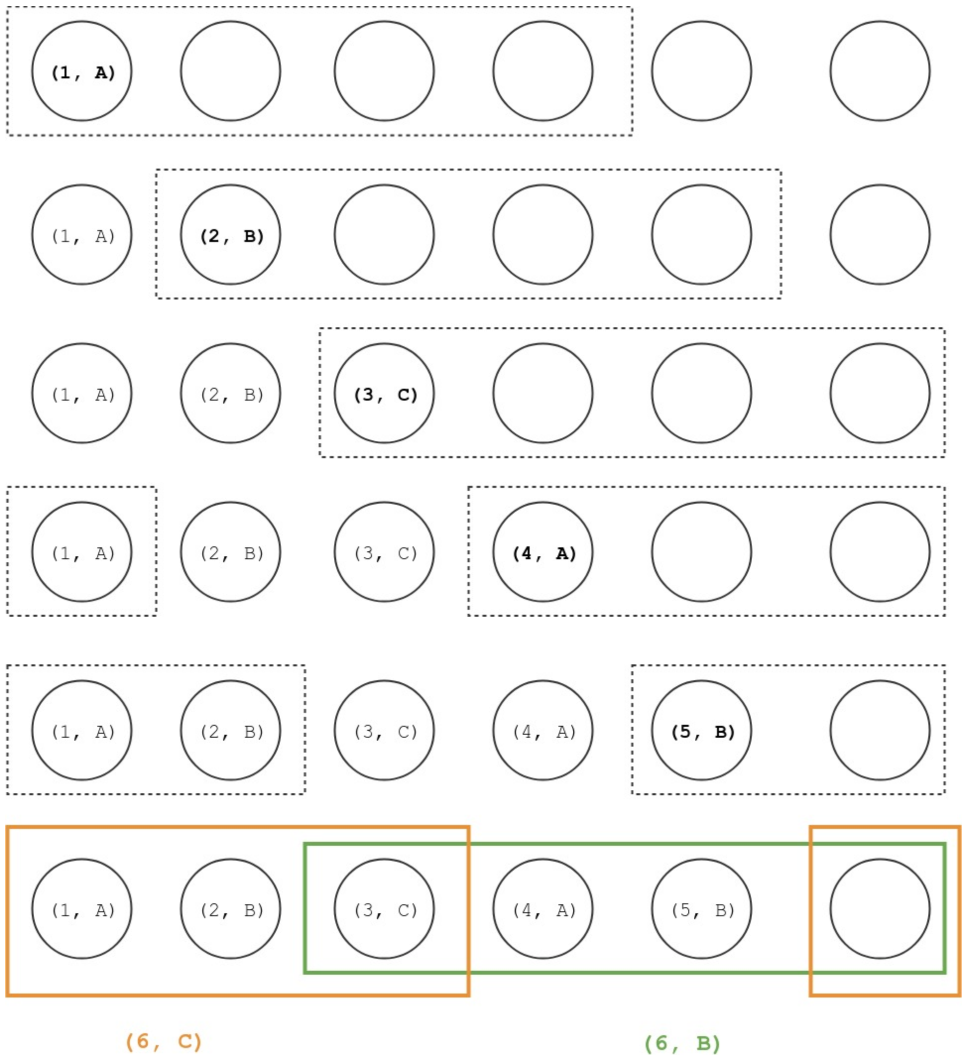


Figure 4.3: A Paxos execution with partial failures

Of course, the clients of the system learn the chosen values, so they could keep track of the state on their side. But, there will always be cases, where some clients need to retrieve some of the values chosen in the past, i.e. because they are clients that have just been brought into operation. So, Paxos should also support *read* operations that return the decisions of previously completed instances alongside *write* operations that start new instances of the protocol. These read operations have to be routed to the current leader of the system, which is essentially the node that completed successfully the last proposal. It's important to note that this node cannot reply to the client using its local copy. The reason for this is that another node might have done a proposal in the meanwhile (becoming the new leader), thus meaning that the reply will not reflect the latest state of the system.[2] As a result, that node will have to perform a read from a majority of nodes, essentially seeing any potential new proposal from another node. You should be able to understand how a majority quorum can guarantee that by now. If not, it would probably be a good idea to revisit the section about quorums and their intersection properties. This means that reads can become quite slow, since they will have to execute in 2 phases. An alternative option that works as an optimisation is to make use of the so-called *master leases* [41]. Using this approach, a node can take a lease, by running a Paxos instance, establishing a point in time,[3] until which it's guaranteed to be considered the leader and no other node can challenge him. This means that this node can then serve read operations locally. However, one has to take clock skew into account in the implementation of this approach and keep in mind it will be safe only if there's an upper bound in the clock skew.

By the same logic, one could argue that electing a leader in every instance of the Paxos protocol is not as efficient as possible and degrades performance significantly under normal conditions without many failures. Indeed, that is true and there is a slightly adjusted implementation of Paxos, called Multi-Paxos that mitigates this issue [42]. In this approach, the node that has performed the last successful proposal is considered the current distinguished proposer. This means that a node can perform a full instance of Paxos and then it can proceed straight to the second phase for the subsequent instances, using the same proposal number that has been accepted previously.

[2]This would mean that the read/write consensus operations would not be *linearizable*. Note that in the context of consensus, operations such as proposals are considered single-object operations. As a result, there is no need for isolation guarantees.

[3]This point in time is essentially the time of the proposal (a timestamp that can be part of the proposal's value) plus a pre-defined time period, which corresponds to *the duration of the lease*.

The rest of the nodes know which node is currently the leader based on which node made the last successful proposal. They can perform periodic health checks and if they believe this node has crashed, they can initiate a `prepare` request in order to perform a successful proposal and become the distinguished proposer. Essentially, this means that the protocol is much more efficient under stable conditions, since it has only one phase. When failures occur, the protocol just falls back to plain Paxos.

Another common need is a way to dynamically update the nodes that are members of the system. The answer to this requirement might sound familiar thanks to the elegance of the protocol; membership information can just be propagated as a new Paxos proposal! The nodes that are member of the system can have their own way of identifying failures of other nodes (i.e. periodic health checks) and the corresponding policies on when a node should be considered dead. When a node is considered dead, one of the nodes that has identified it can trigger a new Paxos instance, proposing a new membership list, which is the previous one minus the dead node. As soon as this proposal completes, all the subsequent instances of Paxos should make use of the updated membership list.

Replicated state machine via consensus

In the beginning of this chapter, we briefly described how a consensus algorithm can be used to solve a wide variety of problems. This is not a coincidence, since all these problems share a common, fundamental characteristic. This is the fact that they can all be modelled as a state machine to some extent. This is also the reason why it's easier to solve them in a centralised setting, but it gets much harder when we want to solve them in a distributed setting in order to increase availability.

However, using a consensus algorithm we can build a *replicated state machine*. This is a set of nodes, where each of them is receiving commands and executing them transitioning between states. If all the nodes make use of the same state machine, then all we need is to make sure that all the nodes receive the same inputs in the same order and then we can guarantee that all the nodes will make exactly the same transitions. This would mean that the distributed system would look similar to a single server from the outside. As a result, one could achieve all the benefits of a distributed system, while maintaining a simple programming model. Figure 4.4 contains a high-level overview of such a system. The top layer is the one receiving requests from the clients and creating proposals for the consensus layer, which conducts the necessary coordination between the other nodes of the system and propagates the chosen values to the lower layer, which just receives these values as inputs and executes the necessary state transitions.

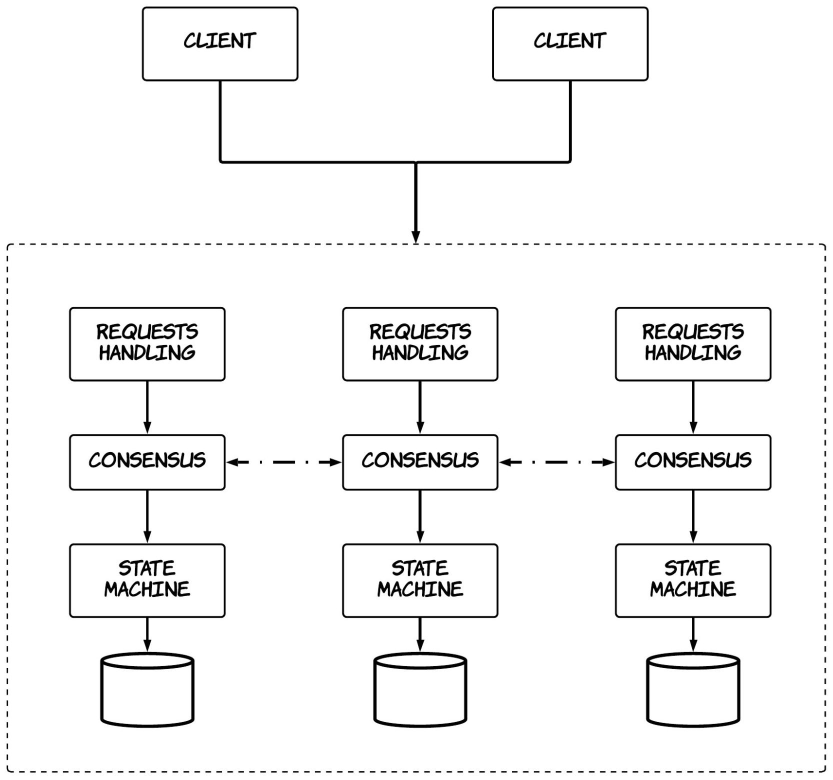


Figure 4.4: A replicated state machine using consensus

Let's elaborate a bit more on what that would entail, assuming Paxos is used as the consensus layer of the system. Essentially, the clients would be sending regular requests to the system, depending on the domain of the system. These requests could be either commands to the system or requests to inspect its internal system. These requests would be dispatched to the current leader of the system, which will be determined based on previous instances of the consensus. If a node that is not a leader receives a request, it can return the leader's address so the client can re-route it. When the system is bootstrapped and no instances of consensus have been run yet, this leader can be determined from a configuration file or the nodes can compete with each other for the leader role. Every time the leader node receives a new command, it attempts to execute a new instance of consensus, increasing the instance number every time. In order to achieve satisfactory performance, multiple consensus instances can be run in parallel. However, the necessary serialization must be performed in some places to ensure correctness. For instance, the lower layer should process the decision of a consensus instance, only when it has processed all the previous instances to ensure that all state machines perform the same transitions. Similarly, the leader should wait after an instance is completed and reply to the associated client only after all the previous instances have completed. During situations where the current leader is unstable and other nodes start making proposals, there might be increased contention for an instance, creating significant delays for any subsequent instances that might have completed. A dummy value can be proposed by the nodes in these cases, which essentially represents a no-op, rejecting the client's operation.

This abstraction of a replicated state machine is quite powerful and could potentially be used to implement solutions for many common problems in the distributed systems area.

Distributed transactions via consensus

The introduction of this chapter mentioned that the consensus problem is very similar to the problem of distributed transactions. However, after studying the Paxos algorithm, one might think there seems to be a fundamental conflict between distributed transactions and the way Paxos solves the consensus problem. The core characteristic of distributed transactions is *atomicity*, the fact that either the relevant update has to be performed in all the nodes or it should not be performed in any of them. However, the Paxos algorithm relies on just a majority quorum to decide on a value. Indeed, the problem of distributed transactions, known as *atomic commit*, and the consensus problem might be closely related, but they are not equivalent[43]. First of all, the consensus problem mandates that every non-faulty node must reach the same decision, while the atomic commit problem requires that all the nodes (faulty or not) must reach the same decision. Furthermore, the atomic commit problem imposes stricter relationships between votes or proposals and the final decision than the consensus problem. In consensus, the only requirement is that the value that is agreed must have been proposed by at least one of the nodes. In atomic commit, a decision can be positive, only if all the votes were positive. The decision is also required to be positive, if all votes are positive and there are no failures.

As a result of this difference, one might think that the Paxos algorithm does not have anything to offer in the problem space of distributed transactions. This is not true and this section will try to illustrate what Paxos (and any other consensus algorithm) has to offer. The biggest contribution of a consensus algorithm would not be in the communication of the *resource managers*' results back to the *transaction manager*, which requires successful communication for all of them and not just a majority. Its value would lie in storing and transmitting the transaction's result back to the resource managers in a fault-tolerant way, so that the failure of a single node (the transaction manager) cannot block the system.

Indeed, there is a very simple way to achieve that in the existing 2-phase commit (2PC) protocol leveraging a consensus algorithm. Assuming we make use of Paxos as a consensus algorithm, we could just have the transaction manager start a new Paxos instance, proposing a value for the result of the transaction, instead of just storing the result locally before sending it back to the resource managers. The proposal value would be either *commit* or *abort*, depending on the previous results of each one of the resource managers. This adjustment on its own would make the 2-phase commit protocol resilient against failures of the transaction manager, since another node could take the role of the transaction manager and complete the protocol. That node would have to read the result of the transaction from any existing Paxos instance. If there's no decision, that node would be free to make an abort proposal.

This is simple and elegant, but it would require adding one more messaging round to the 2-phase commit protocol. It's actually possible to remove this additional round, trading off some simplicity for increased performance. This could be done by essentially "*weaving*" several instances of Paxos in the plain 2-phase commit protocol, practically obviating the need for a transaction manager completely. More specifically, the resource managers would have to send their response to the first phase to a set of acceptors, instead of sending it to the transaction manager, thus creating a separate Paxos instance for every resource manager involved in the transaction. Similarly, the acceptors could propagate the chosen values to the resource managers directly, instead of doing that indirectly via the transaction manager. The resource managers would be responsible for checking that all the Paxos instances from the other resource managers had a positive result (corresponding to the first phase of 2PC) in order to commit the transaction.

A paper titled 'Consensus on transaction commit' [44] examines this relationship between distributed transactions and consensus and also explains this approach in much greater detail, which is referred to as *Paxos commit*. This paper also demonstrates why 2-phase commit is essentially a special case of Paxos commit with zero tolerance of node failures (`f = 0`).

Raft

Paxos has been the canonical solution to the consensus problem. However, the initial specification of the algorithm did not cover some aspects that were crucial in implementing the algorithm in practice. As explained previously, some of these aspects were covered in subsequent papers. The Paxos algorithm is also known to be hard to understand.

As a response to these issues, researchers decided to create a new algorithm with the goals of improved understandability and ease of implementation. This algorithm is called Raft [45]. We will briefly examine this algorithm in this section, since it has provided a good foundation for many practical implementations and it nicely demonstrates how the various aspects described before can be consolidated in a single protocol.

Raft establishes the concept of a replicated state machine and the associated replicated log of commands as first class citizens and supports by default multiple consecutive rounds of consensus. It requires a set of nodes that form the consensus group, which is referred to as the *Raft cluster*. Each of these nodes can be in one of 3 states: a leader, a follower or a candidate. One of the nodes is elected the leader and is responsible for receiving log entries from clients (proposals) and replicate them to the other follower nodes in order to reach consensus. The leader is responsible for sending heartbeats to the other nodes in order to maintain its leadership. Any node that hasn't heard from the leader for a while will assume the leader has crashed, it will enter the candidate state and attempt to become leader by triggering a new election. On the other hand, if a previous leader identifies another node has gained leadership, it falls back to a follower state. Figure 4.5 illustrates the behaviour of the nodes depending on their state.

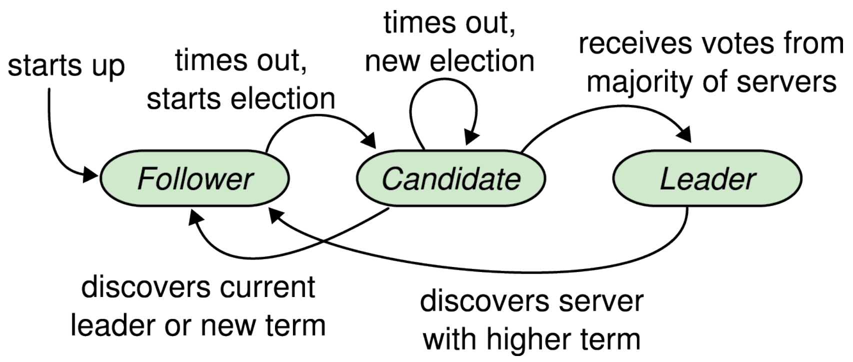

starts up
times out, starts election
times out, new election
receives votes from majority of servers
Follower
Candidate
Leader
discovers current leader or new term
discovers server with higher term

Figure 4.5: The states of nodes in Raft

In order to prevent two leaders from operating concurrently, Raft has the temporal concept of *terms*. Time is divided into terms, which are numbered with consecutive integers and each term begins with an election where one or more candidates attempt to become leaders. In order to become a leader, a candidate needs to receive votes from a majority of nodes. Each node votes for at most one node per term on a first-come-first-served basis. Consequently, at most one node can win the election for a specific term, since 2 different majorities would conflict in at least one node. If a candidate wins the election, it serves as the leader for the rest of the term. Any leader from previous terms will not be able to replicate any new log entries across the group, since the voters of the new leader will be rejecting its requests and it will eventually discover it has been deposed. If none of the candidates manages to get a majority of votes in a term, then this term ends with no leader and a new term (with a new election) begins straight after.

Nodes communicate via remote procedure calls (RPCs) and Raft has 2 basic RPC types:

- `RequestVote`: sent by candidates during an election.
- `AppendEntries`: sent by leaders to replicate log entries and also to provide a form of heartbeat.

The commands are stored in a log replicated to all the nodes of the cluster. The entries of the log are numbered sequentially and they contain the term in which they were created and the associated command for the state machine, as shown in Figure 4.6. An entry is considered *committed* if it can be applied to the state machine of the nodes. Raft guarantees that committed entries are durable and will be eventually be executed by all of the available state machines, while also guaranteeing that no other entry will be committed for the same index. It also guarantees that all the preceding entries of a committed entry are also committed. This status essentially signals that consensus has been reached on this entry.

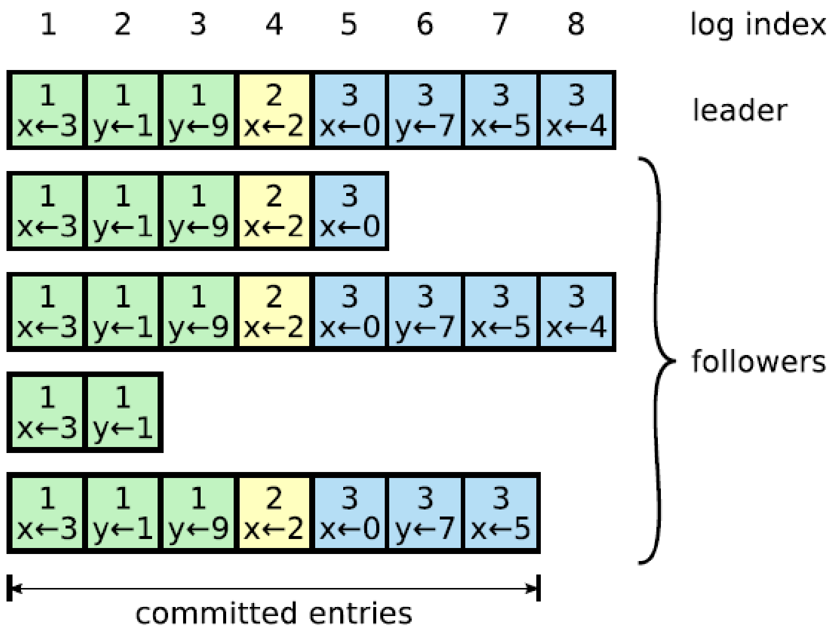


Figure 4.6: The structure of the replicated log

As mentioned previously, leaders are responsible for receiving commands from clients and replicating them across the clusters. This happens in the following order: when a leader receives a new command, it appends the entry to its own log and then sends an `AppendEntries` request in parallel to the other nodes, retrying when it does not receive a timely response. When the leader receives a response from a majority of followers, the entry can be considered committed. The leader applies the command to its state machine and informs the followers they can do the same, by piggybacking the necessary information about committed entries in subsequent `AppendEntries` messages. Of course, this is mostly the happy path.

During leader and follower failures, divergence might be observed between the various nodes. Figure 4.7 contains some examples of this phenomenon.

For example, a follower might have crashed and thus missed missed some (committed) entries (a, b). It might have received some more (non committed) entries (c,d). Or both things might have happened (e,f). Specifically scenario (f) could happen if a node was elected leader in both terms 2 and 3 and replicated some entries, but it crashed before any of these entries were committed.

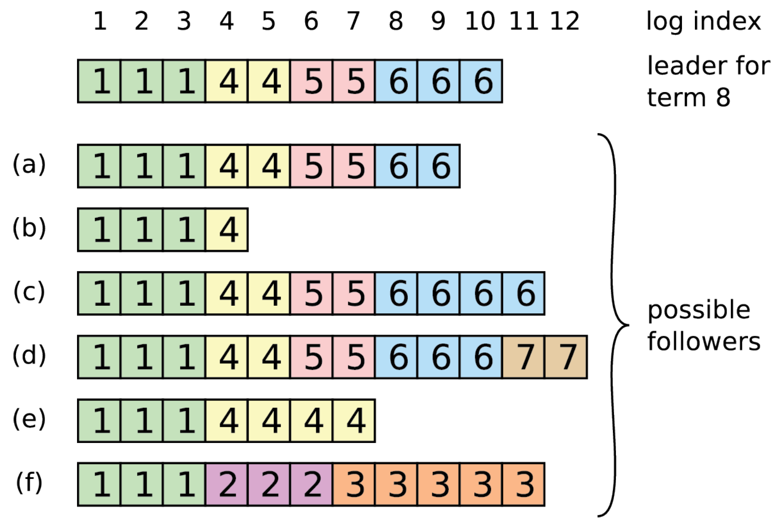


Figure 4.7: Temporary divergence of node logs

Raft contains some additional elements to resolve these temporary divergences. The main overarching principle is that any elected leader should contain any entries that have been committed up to the term he becomes leader. The leader is then responsible for helping any followers with conflicting logs adjust them accordingly to converge again. It's important to note that a leader only appends entries to its log and never updates it, only followers are allowed to update their log. These 2 aspects are satisfied in the following way:

- During an election, every `RequestVote` RPC contains some information about the candidate's log. A voter is allowed to vote for a candidate only if its log is not more *up-to-date* than the candidate's log. Raft determines which of two logs is more up-to-date by comparing the index and term of the last entries in the logs. A candidate must receive votes from a majority of the cluster in order to be elected, which means that every committed entry must be present in at least one of those servers. If the candidate's log is at least as up-to-date as any other log

in that majority, then it's guaranteed to hold all the committed entries.
- When sending an `AppendEntries` RPC, a leader includes the index and term of the entry that immediately precedes the new entries in its log. The followers check against their own logs and reject the request if their log differs. If that happens, the leader discovers the first index where their logs disagree and starts sending all the entries after that point from its log. The follower discards its own entries and adds the leader's entries to its log. As a result, their logs eventually converge again.

We mentioned previously that a leader knows that an entry from its term can be considered committed when it has been successfully replicated to a majority of nodes and it can then be safely applied to the state machine. But, what happens when a leader crashes before committing an entry? If subsequent leaders have received this entry, they will attempt to finish replicating the entry. However, a subsequent leader cannot safely conclude that an entry from a previous term is commmitted once it is stored on a majority of nodes. The reason is there is an edge case where future leaders can still replace this entry even if it's stored on a majority of nodes. Feel free to refer to the paper for a full description of how this can happen. As a result, leaders can safely conclude an entry from a previous term is committed by replicating it and then replicating a new entry from its term on top of it. If the new entry from its own term is replicated to a majority, the leader can safely consider it as committed and thus it can also consider all the previous entries as committed at this point. So, a leader is guaranteed to have all the committed entries at the start of its term, but it doesn't know which those are. To find out, it needs to commit an entry from its own term. To expedite this in periods of idleness, the leader can just commit a no-op command in the beginning of its term.

What has been described so far consists the main specification of the Raft protocol. The paper contains more information on some other implementation details that will be covered briefly here. Cluster membership changes can be performed using the same mechanisms by storing the members of the cluster in the same way regular data is stored. An important note is that transition from an old configuration C_{old} to a new configuration C_{new} must be done via a transition to an intermediate configuration C_{joint} that contains both the old and the new configuration. This is to prevent two different leaders from being elected for the same term. Figure 4.8 illustrates how that could happen if the cluster transitioned from C_{old} directly to C_{new}. During this intermediate transition, log entries are replicated to the servers of both configurations, any node from both configurations can serve as a leader and consensus requires majority from both the old and the new configuration. After the C_{joint} configuration has been committed, the cluster then switches to the new configuration C_{new}. Since the log can grow infinitely, there also needs to be a mechanism to avoid running out of storage. Nodes can perform log compaction by writing a snapshot of the current state of the system on stable storage and removing old entries. When handling read requests from clients, a leader needs to first send heartbeats to ensure it's still the current leader. That guarantees the linearizability of reads. Alternatively, leaders can rely on the heartbeat mechanism to provide some form of lease, but this would assume bounded clock skew in order to be safe. A leader might also fail after applying a committed entry to its state machine, but before replying to the client. In these cases, clients are supposed to retry the request to the new leader. If these requests are tagged with unique serial numbers, the Raft nodes can identify commands that have already been executed and reply to the clients without re-executing the same request twice.

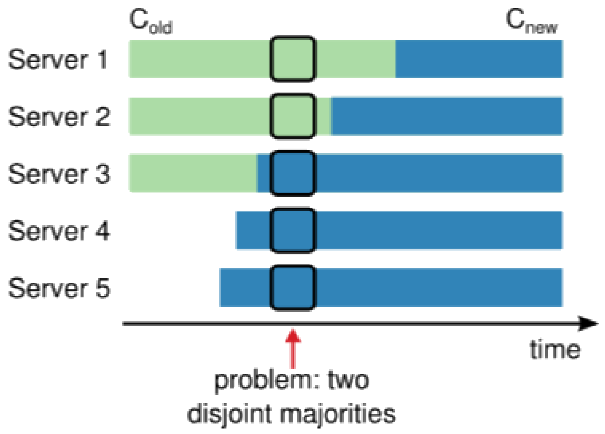


Figure 4.8: Risk of switching directly from C_{old} to C_{new}

Standing on the shoulders of giants

At this point, we have spent enough time examining all the small details of the various consensus algorithms. This can prove to be very useful, when one thinks about how to design a system, what kinds of guarantees it would require or even when troubleshooting edge cases. Hopefully, you have realised by now that these problems are very complicated. As a result, creating an algorithm that solves these problems or even translating an existing algorithm to a concrete implementation is a really big undertaking. If there is an existing solution out there, you should first consider re-using this before rolling out your own, since it's highly likely that the existing solution would be much more mature and battle-tested. This is true not only for consensus but other problems inherent to distributed systems as well. A later chapter contains some case studies of basic categories of such distributed systems that you can leverage to solve some common problems.

Part III

Time & Order

Time and *order* are some of the most challenging aspects of distributed systems. As you might have realised already by the examples presented, they are also very intertwined. For instance, time can be defined by the order of some events, such as the ticks of a clock. At the same time, the order of some events can also be defined based on the time each of these events happened. This relationship can feel natural when dealing with a system that is composed of a single node, but it gets a lot more complicated when dealing with distributed systems that consist of many nodes. As a result, people that have been building single-node, centralised applications sometimes get accustomed to operating under principles and assumptions that do not hold when working in a distributed setting. This part will study this relationship between time and order, the intricacies related to distributed systems and some of the techniques that can be used to tackle some of the problems inherent in distributed systems.

Chapter 5

Time

What is different in a distributed system

As mentioned previously, one of the main use cases of time in a software system is to determine the order between different events. In practice, this can mean very different things. For example, a system might want to impose an order to requests received concurrently by different clients in order to determine in which order the effects of each request should take place. A different example is one where a system administrator might be investigating an incident looking at the system logs and trying to infer relationships between different events, using the timestamp of the logs associated to these events. Both of these examples have a common, underlying goal, which is to determine the order between events.

There is a main difference between a centralised system and a distributed, multi-node system with regards to time. In the first type of systems, there is only a single node and thus only a single clock. This means one can maintain the illusion of a single, universal time dimension, which can determine the order of the various events in the single node of the system. In a distributed system, each node has its own clock and each one of those clocks may run at a different rate or granularity, which means they will drift apart from each other. As a consequence of this, **in a distributed system there is no global clock**, which could have been used to order events happening on different nodes of the system.

A practical perspective

The clocks used in real systems are what we usually call **physical clocks**. A physical clock is a physical process coupled with a method of measuring that process to record the passage of time [46]. Most physical clocks are based on cyclic processes. Below are some examples of such devices:

- Some of the most basic ones and easy to understand are devices like a *sundial* or an *hourglass*. The former tells the time of the day using a gnomon and tracking the shadow created by the sun. The latter measures time by the regulated flow of of sand through a bulb.
- Another common clock device is a *pendulum clock*, which uses an oscillating weight as its timekeeping element.
- An electronical version of the last type is used in software systems, called a *quartz clock*. This device makes use of a crystal, called quartz crystal, which vibrates or ticks with a specific frequency, when electricity is applied to it.
- One of the most accurate timekeeping devices are *atomic clocks*, which use the frequency of eletronic transitions in certain atoms to measure time.

As explained initially, all these devices rely on physical processes to measure time. Of course, there can be errors residing both in the measurement tools being used and the actual physical processes themselves. As a result, no matter how often we synchronize these clocks with each other or with other clocks that have more accurate measurement methods, there will always be a skew between the various clocks involved in a distributed system. When building a distributed system, this difference between clocks must be taken into account and the overall system should not operate under the assumption that all these clocks are the same and can act as a single, global clock.

Figure 5.1 contains an example of what could happen otherwise. Let's assume we have a distributed system composed of 3 different nodes A, B and C. Every time an event happens at a node, the node assigns a timestamp to the event, using its own clock, and then propagates this event to the other nodes. As the nodes receive events from the other nodes, they compare the timestamps associated with these events to determine the order in which the events happened. If all the clocks were completely accurate and reflecting exactly the same time, then that scheme would theoretically be capable of identifying the order. However, if there is a skew between the clocks of the various nodes, the correctness of the system is violated. More specifically, in our example, we assume that the clock of node A is running ahead of the clock of node B. In the same way, the clock of node C is running behind the clock of node B. As a result, even if the event in node A happened before the event in node C, node B will compare the associated timestamps and will believe the event from node C happened first.

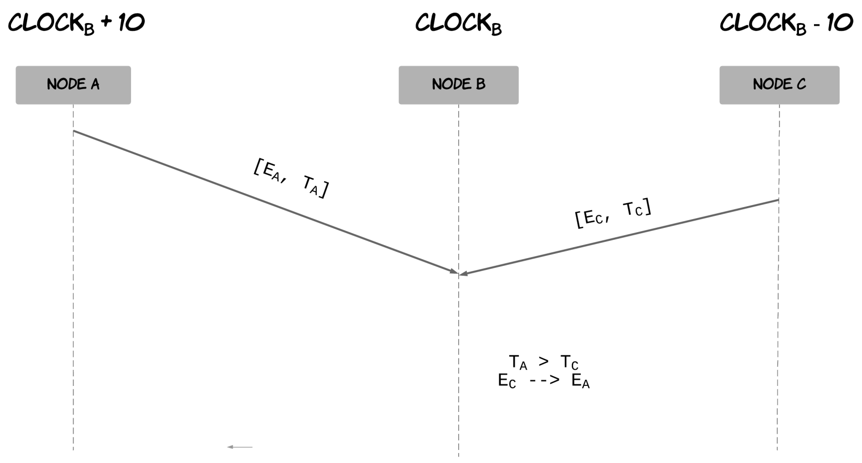


Figure 5.1: Side-effects of assuming a global clock

So, from a practical point of view, the best we could do is accept there will always be a difference between the clocks of different nodes in the system and expose this uncertainty in some way, so that the various nodes in the system can handle it appropriately. Spanner [5] is a system that follows this approach, using the TrueTime API that directly exposes this uncertainty by using time intervals (embedding an error) instead of plain timestamps.

A theoretical perspective

What we have demonstrated so far is a rather practical and simplified version of the notion of time. The main hypothesis was that there exists an absolute, universal notion of time, but it's practically impossible to measure it in a completely precise way. In reality, there is no such absolute and universal notion of time. According to the laws of physics, more specifically the *special* *theory of relativity*,[1] it is impossible to say in an absolute sense that two distinct events occur at the same time if those events are separated in space. For example, a car crash in London and another in New York appearing to happen at the same time to an observer on Earth, will appear to have occurred at slightly different times to an observer on an airplane flying between London and New York.

This can look like a paradox, but you can easily understand the intuition behind this theory via a thought experiment, known as the *train experiment*. It consists of a train with an observer located in the middle of the carriage and another observer standing on the platform as the train passes by. A flash of light is given in the middle of the carriage right at the point when the two observers meet each other. For the observer inside the carriage, both sides of the carriage have the same distance from the middle, so the light will reach both sides at the same time. For the observer standing on the platform, the rear side is moving towards the initial point of the flash, while the front side is moving away from it, so the light will reach the two sides at different points in time, since the speed of light is the same for all directions. As a result, **whether two spatially separated events happen at the same time (simultaneously) is not absolute, but depends on the observer's reference frame**.

Note that in practice relativity is not a problem for computing systems, since all of humanity's current computing computing systems share a close enough frame of reference to make relativistic differences in the perception of time immaterial. In the context of this book though, it mainly serves to underline the fact that time is relative and there can be no absolute, universal time dimension in a distributed system. It also gives rise to another observation that forms the basis of distributed algorithms, which provide solutions to ordering problems without relying on a global clock. These algorithms will be covered in more detail in the next chapter. As illustrated from the previous example, information in the real world flows through light. If the sun stops shining, a human will realise that a bit later (around 8 minutes later, to be concrete). This is the time it takes sunlight to reach Earth from the sun. If you think about it, distributed systems are very similar. Events happen at different nodes of the system and it takes some time for the information that these events happened to propagate to the other nodes of the system. A difference worth pointing out is that the speed of light is constant, but the speed with which information flows in a distributed system is variable, since

[1] See: https://en.wikipedia.org/wiki/Relativity_of_simultaneity

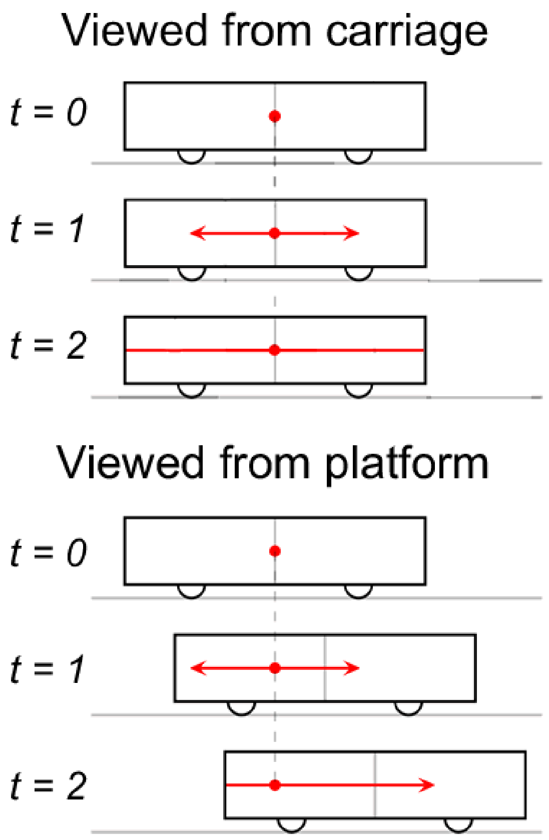


Figure 5.2: The train experiment

it depends on the underlying infrastructure and conditions of the network. In the worst-case scenario, information might not be able to flow between 2 parts of a system at all because of a network partition.

Logical clocks

The focus of this section has been on physical clocks so far, explaining their main limitations when used in distributed systems. However, there is an alternative category of clocks, which is not subject to the same constraints, **logical clocks**. These are clocks that do not rely on physical processes to keep track of time. Instead, they make use of messages exchanged between the nodes of the system, which is the main mechanism information flows in a distributed system, as described previously.

We can imagine a trivial form of such a clock in a system consisting of only a single node. Instead of using a physical clock, this node could instead make use of a logical clock, which would consist of a single method, say `getTime()`. When invoked, this method would return a counter, which would subsequently be incremented. For example, if the system started at 9:00 and events A, B and C happened at 9:01, 9:05 and 9:55 respectively, then they could be assigned the timestamps 1, 2 and 3. As a result, the system would still be able to order the events, but it would not be able to determine the temporal distance between any two events. Some more elaborate types of logical clocks are described in the next section.

Chapter 6

Order

Total and partial ordering

As explained before, determining the order of events is a common problem that needs to be solved in software systems. However, there are 2 different, possible types of ordering: *total ordering* and *partial ordering*.

Total order is a binary relation that can be used to compare any 2 elements of a set with each other. As a direct consequence of this property, using this relation we can derive only a single order for all the elements in a set, which is why it's called total order. For instance, the set of unique integers `{7, 9, 3, 2, 6}` can be totally totally ordered (using the relation less than `<`) and the associated total order is `[2, 3, 6, 7, 9]`.

Partial order is a binary relation that can be used to compare *only* some of the elements of a set with each other. As a consequence, using this relation we can derive multiple, valid orders for all the elements in the set. For example, the set of the following sets of integers `{{0}, {1}, {2}, {0,1}, {1,2}}` can only be partially ordered, using the subset relation $\subseteq$. If we pick the elements `{0}` and `{0,1}`, then they can be ordered, since `{0}` $\subseteq$ `{0,1}`. However, the elements `{0,1}` and `{1,2}` cannot be ordered using this relation, since neither `{0,1}` $\subseteq$ `{1,2}` nor `{1,2}` $\subseteq$ `{0,1}` holds. As a result, both `[{0}, {1}, {2}, {0,1}, {1,2}]` and `[{2}, {1}, {0}, {1,2}, {0,1}]` would be valid partial orderings.

As the previous chapter implied, in systems that are composed of a single node, it's easy and intuitive to determine a total ordering of events. The main reason is that there is a single actor, where all the events happen, so this actor can impose a total order on these events as they occur. Total orderings also make it much simpler to build protocols and algorithms. However, in a distributed system it's not that straightforward to impose a total order on events, since there are multiple nodes in the system and events might be happening concurrently on different nodes. As a result, a distributed system can make use of any valid partial ordering of the events occuring, if there is no strict need for a total ordering.

Figure 6.1 contains a diagram that illustrates why total ordering is much harder to determine in a distributed system. As displayed in the diagram, in a system composed of a single node that can only execute events serially, it's easy to define a total order on all the events happening, since between two events (e_1, e_2) one of them will have started after the other one finished. On the other hand, in a distributed system composed of multiple nodes where events are happening concurrently, it's much harder to determine a total order, since there might be pairs of events that cannot be ordered.

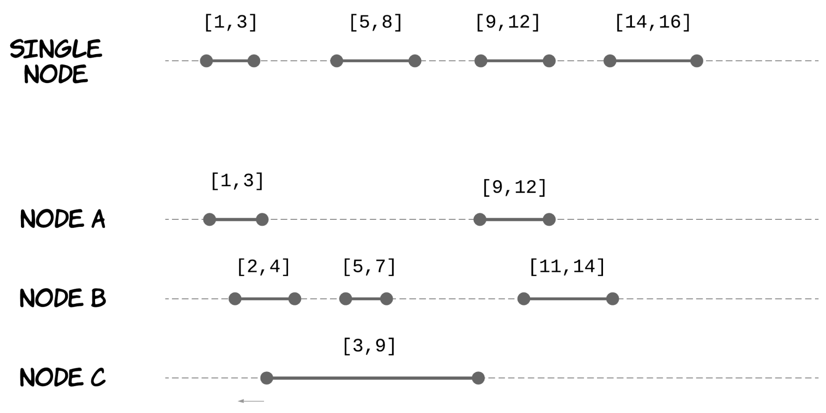


Figure 6.1: Total ordering vs partial ordering

An interesting and important observation can be made at this point: the fact that these operations have some duration and are interleaved with each other is not the only problem that makes a total ordering hard to achieve. Even if these operations are instantaneous (also referred to as linearizable), then total ordering is still non-trivial to achieve, since there is no global clock, as we explained previously. As a result, even if each node can assign unique timestamps to the events happening, these timestamps will be coming from clocks running at different rates, thus making it harder to compare them. This is demonstrated in Figure 6.2, which shows that in a single-node system, any clock errors can be ignored, since there is only one clock in use. This makes it possible to assign timestamps to events as if they are instantaneous and establish a total order. However, in a distributed system, there are multiple clocks in play that that run in different rates and can have different errors. This means that the errors need to be taken into account, when comparing timestamps between different nodes, thus making it harder to establish a total order, since there will be pairs of events where one cannot know which of them happened first. Note that for ease of understanding, in the figure the clocks of all the nodes appear to have the same error, which is not realistic.

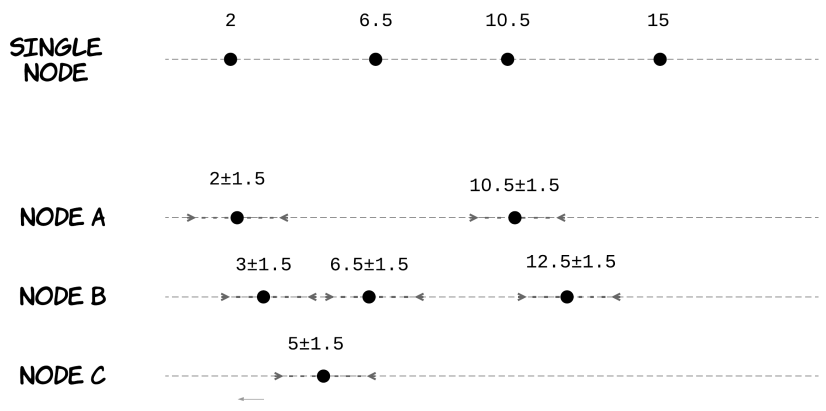


Figure 6.2: Clock errors making total ordering harder

The concept of causality

As humans, we grow accustomed to total ordering, because most of the natural phenomena around us appear to be subject to total ordering. When we go shopping, we are placed in a queue, so that we can be served in a total order. Similarly, cars waiting for the signal to change at an intersection are also ordered in the same way. However, there are scenarios - especially prevalent in software systems - where a total ordering is not really necessary.

For instance, look at some of the social media platforms people use nowadays, where they can create posts and add comments to the posts of other people. Do you really care about the order in which two unrelated posts are shown to you? Probably not. As a result, the system could potentially leverage a partial ordering, where posts that can't really be ordered are displayed in an arbitrarily chosen order. However, there is still a need for preserving the order of some events that are tightly linked. For example, if a comment C_B is a reply to a comment C_A, then you would most probably like to see C_B after C_A. Otherwise, a conversation could end up being confusing and hard to follow.

What we just described is the notion of **causality**, where one event contributes to the production of another event. Looking back at one of the introductory sections, you can find the description of a consistency model, called the *causal consistency model*, which ensures that events that are causally related are observed by the various nodes in a single order, where causes precede the effects. Violating causality can lead to behaviours that are really hard to understand by the users of a system. Fortunately, as we will explore in the next sections of this chapter, it's possible to track causality without the need of physical time.

The notion of causality is also present in real life. We subconsciously make use of causality when planning or determining the feasibility of a plan or the innocence of an accused.[1] Causality is determined based on a set of loosely synchronized clocks (i.e. wrist watches, wall clocks etc.) under the illusion of a global clock. This appears to work in most cases, because the time duration of events is much more coarse-grained in real life and information *"flows"* much more slowly than in software systems. For instance, compare the time a human needs to go from London to Manchester and the time needed for 10 kilobytes to travel the same distance via the Internet. As a result, small differences between clocks do not create significant problems in most real life scenarios. However, in distributed computing systems, events happen at a much higher rate, higher speed and their duration is several orders of magnitude smaller. As a consequence, if the physical clocks of the various nodes in the system are not precisely synchronised, the causality relation between events may not be accurately captured.

To sum up, causality can be leveraged in the design of distributed systems with 2 main benefits: increasing concurrency and replacing real time with the notion of logical time, which can be tracked with less infrastructure and costs.

[1]See: https://en.wikipedia.org/wiki/Alibi

As we have seen so far, distributed systems are inherently asynchronous. By introducing coordination and synchronisation between them, we essentially reduce the level of concurrency and consequently their performance. The notion of causality allows us to allow these systems to remain asynchronous while also supporting the causal consistency model, which prevents a big set of counter-intuitive behaviours which stem from weak consistency. Furthermore, keeping physical clocks synchronised is a task that requires hardware infrastructure with the associated costs, where the costs increase significantly the more accurate the synchronisation needs to be. Logical clocks rely on the existing messaging exchanged between nodes of a system, which makes them less expensive to implement. Of course, logical clocks have their own pitfalls, as some of the next sections will explain, so they are definitely not a silver bullet.

As implied already, it's possible to track causality relationships between events without using physical clocks, using *logical clocks* instead. The following sections will present some kinds of logical clocks, but it's important to highlight in advance that they all share some common characteristics [47]. The abstraction of logical clocks consist of 2 main parts:

- a data structure local to every node used to represent logical time.
- a protocol to update the data structures accordingly as events happen and time passes by.

Each node maintains data structures that provide the following capabilities:

- a *local logical clock* that helps a node measure its own progress.
- a *global logical clock* that is a good representation of a node's view of the logical global time.

Similarly, the protocol consists of 2 main rules:

- R1: a rule that governs how the local logical clock is updated by a node when it executes an event.
- R2: a rule that governs how a node updates its global logical clock to update its view of the global time and progress. This determines what information about the logical time needs to be piggybacked in the messages exchanged and how this information is used by the receiving node.

Different types of logical clocks have the same core parts and the protocol described above, but they might differ in the actual data structures used to represent logical time or the logic in the rules of the protocol.

The events that happen in a distributed system can be classified in 3 very basic categories: local events happening at a node and changing its state, send events that represent a node sending a message to another node to inform about a change and receive events that represent a node receiving a message from another node about a change. As implied before, these events that are exchanged between nodes in order to propagate information can also propagate time changes between nodes. The notion of causality is built on top of the **happened-before**[2] relation ($\rightarrow$). This is a strict, partial order on the aforementioned events so that:

- If events a and b are two events happening at the same node, the relation a $\rightarrow$ b holds if the occurence of event a preceded the occurence of event b. Note that this is easy to determine for a single node, as shown before.
- If event a is the event of a node corresponding to sending a message and event b is the event of a different node corresponding to the receipt of the same message, then a $\rightarrow$ b.
- For 3 events a, b, c, if a $\rightarrow$ b and b $\rightarrow$ c, then a $\rightarrow$ c.

We can say that event e_1 causally precedes event e_2 (or these 2 events are *causally related*) if $e_1 \rightarrow e_2$. We can say that event e_1 is not causally related to e_2 ($e_1 \| e_2$), if none of the relations $e_1 \rightarrow e_2$ or $e_2 \rightarrow e_1$ holds.

Figure 6.3 demonstrates how this would work in a distributed system of 3 nodes. Applying the rules described above, one can see the following causal relationships: $e_1 \rightarrow e_4$, $e_1 \rightarrow e_5$, $e_1 \rightarrow e_6$, $e_1 \rightarrow e_7$, $e_1 \rightarrow e_8$ and $e_1 \rightarrow e_9$. However, note that $e_1 \| e_3$ and $e_2 \| e_6$ even though these events are temporally distant, because there was no information exchanged between the nodes that could help the logical clocks track any relation between them. This means that these pair of events should be considered as concurrent by the system and could have happened in any order. The reason for this is that e_6 could have happened at any point in time from the moment right after e_1 to just before e_9 preserving all the causal relationships intact. The same holds true for e_2, which could have happened at any point in time until e_4.

In general, the concept of causality in distributed systems (and as defined in the Lamport paper[48]) could be referred to as *potential causality*, since it does not necessarily indicate a cause-and-effect relationship, but only a potential for it. As a result, when we say that $e_i \rightarrow e_j$, we don't mean that

[2]See: https://en.wikipedia.org/wiki/Happened-before

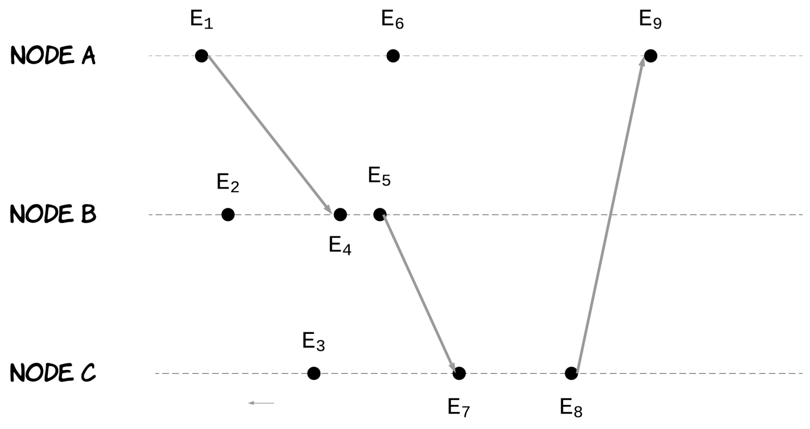


Figure 6.3: Happened-before relationship in a distributed system

e_i has caused or affected e_j. Instead, we mean that e_i *could* have caused or affected e_j. This is because this concept is generic and does not have application-specific context to infer actual cause-and-effect relationships. However, applications can leverage the algorithms presented in this chapter, enriching them with additional metadata that make it possible to track *actual causality*, instead of *potential causality*. Note that even the ability to track potential causality is still very useful to prevent system behaviours that will be confusing to users, it just means that the system might be storing and transmitting more information than necessary to achieve this purpose.

Lamport clocks

One of the first and simplest types of logical clocks were invented by Leslie Lamport and is called **Lamport clock** [48]. In this type of logical clock, every node in the system maintains a logical clock in the form of a numeric counter that can start from zero when a node starts operating. The rules of the protocol are the following:

- (R1) Before executing an event (send, receive or local), a node increments the counter of its logical clock by one: $C_i = C_i + 1$.
- (R2) Every sent message piggybacks the clock value of its sender at sending time. When a node n_i receives a message with timestamp C_{msg},

it executes the following actions:

- Updates its clock by taking the maximum of its clock and the received clock: $C_i = \max(C_i, C_{msg})$
- Executes R1
- Delivers the message

The Lamport clocks satisfy the so-called **clock consistency condition**: if one event e_1 causally precedes another event e_2, then $C(e_1) < C(e_2)$. However, the reverse which is known as the **strong consistency condition** is not satisfied by Lamport clocks. This means that if $C(e_1) < C(e_2)$, then this does not mean necessarily that the event e_1 causally precedes e_2. As a result, this means that Lamport clocks cannot be used to infer partial orderings that are causally consistent. However, they can still be used for other purposes, such as creating (non causally consistent) total orderings, as we will explain later.

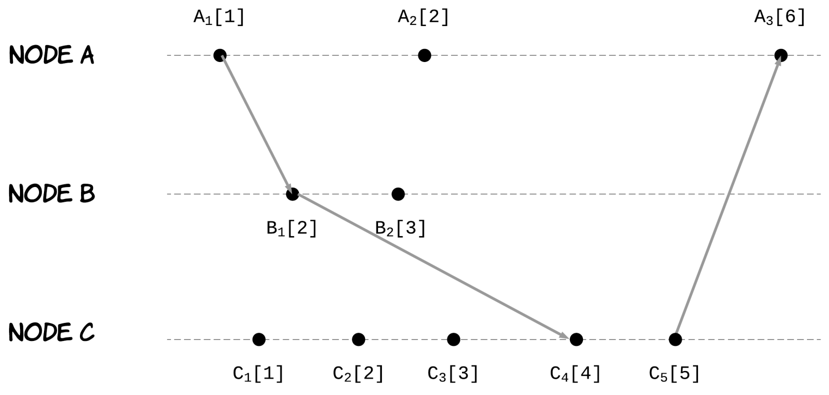


Figure 6.4: Lamport clocks

To understand better how Lamport clocks work, let's look at an example, as shown in Figure 6.4. We have a distributed system that consists of 3 nodes A, B and C that execute events locally and exchange messages to propagate the necessary information across the whole system. You can try running the rules described above and see that the clocks of each node will be updated as shown in the figure. Essentially, each node ticks its clock as local events happen and also bumps the clock in case it identifies that another node has a higher clock value to that node's value. Now, let's discuss the conditions presented previously a bit more. Any two events that are causally related will reflect this relationship in the clock's value. For instance, A_1 causally precedes B_1 and we can see that $C(A_1) = 1 < 2 = C(B_1)$ (clock consistency condition). We can also see that the strong consistency condition does not hold. For instance, $C(C_2) < C(B_2)$, but these 2 events are not causally dependent. Event B_2 could have happened either before or after C_2 with the same clock value.

Lamport clocks can be used to create a total ordering of events in a distributed system by using some arbitrary mechanism to break ties, in case clocks of different nodes have the same value (e.g. the ID of the node). The caveat is this total ordering is somewhat arbitrary and cannot be used to infer causal relationship, which limits the number of practical applications they can have. The paper demonstrates how they could potentially be used to solve synchronisation problems, such as mutual exclusion[48].

Vector clocks

As explained in the previous section, the main limitation of Lamport clocks is that they do not satisfy the strong clock condition, which means they cannot be used to infer causal relationships between events. The main underlying reason for this is the fact that both the local and the global logical clocks for each node are flattened into a single number, which does not provide all the necessary information in order to track causal relationships. What we need to do essentially is maintain for each event a set of all events that causally precede it, which is known as a **causal history**[49]. For instance, in Figure 6.3 the causal history of e_7 is $\{e_1, e_2, e_3, e_4, e_5\}$. We also need to store the causal history of each event as efficiently as possible, using a compact data structure. A **vector clock** is an example of such a data structure[50][51].

A vector clock is another type of logical clock, where the clock data structure for each node is a vector of N counters (where N is the number of nodes in the system) $[c_0, c_1, .., c_N]$. For the clock of the i^{th} node $[c_{i,0}, c_{i,1}, ..., c_{i,N}]$:

- the i^{th} element of the clock $c_{i,i}$ represents the *local logical clock* of the node.
- the remaining elements of the clock $[c_{1,0}, ..., c_{i,i-1}, c_{i,i+1},..., c_{1,N}]$ form together the *global logical clock* of the node.

The rules of the protocol are the following:

- (R1) Before executing an event (send, receive or local), a node increments the counter of its logical clock by one: $C_{i,i} = C_{i,i} + 1$.
- (R2) Every sent message piggybacks the clock value of its sender at sending time. When the i^{th} node receives a message with a vector $[c_{j,0}, ..., c_{j,N}]$ from the j^{th} node, it executes the following actions:
  - Executes R1
  - Updates each element in its vector by taking the maximum of the value in its own vector clock and the value in the vector in the received message: $C_{i,k} = max(c_{i,k}, c_{j,k})$ for every k in [0, N]
  - Delivers the message

This type of clock satisfies the *strong clock condition*. This means that if for 2 events e_i, e_j with timestamps C_i, C_j the relationship $C_i < C_j$ holds, then $e_i \rightarrow e_j$. The only missing thing is how to compare vector clocks with each other, which is done in the following way:

- For 2 clocks $C_i = [c_{i,0}, ..., c_{i,N}]$ and $C_j = [c_{j,0}, ..., c_{j,N}]$ $C_i < C_j$ iff all the elements of the clock C_i are less than or equal to all the corresponding elements of clock C_j ($c_{i,k} \leq c_{j,k}$ for all k) and there is at least one element of C_i that is strictly smaller than the corresponding element of C_j ($c_{i,l} < c_{j,l}$ for at least one l in [0, N]).

Figure 6.5 contains the same distributed execution displayed in Figure 6.4, but using vector clocks this time. Each node in the system maintains a vector clock, where the first element corresponds to the time in node A, the second to the time in node B and the third and last element to the time in node C. You can spend some time again to verify that the clock values have been assigned just by following the rules described above.This time, we can see that the clock of A_1 is smaller than the clock of B_1, while also $A_1 \rightarrow B_1$. What's more important though is that we can detect events that are not causally related. For instance, $B_2 \parallel C_2$ and we can see that the timestamp of B_2 is neither smaller or larger than the timestamp of C_2. This means that we can consider these events as concurrent and they could have happened in any order.

Vector clocks can be used in cases that benefit from the capability to detect whether two events are causally related or concurrent, while allowing the different nodes of the system to make progress independently and efficiently without synchronisation and coordination bottlenecks.

We mentioned previously that in a distributed system of `n` nodes, each vector clock is composed of n elements. It's important to clarify that every process

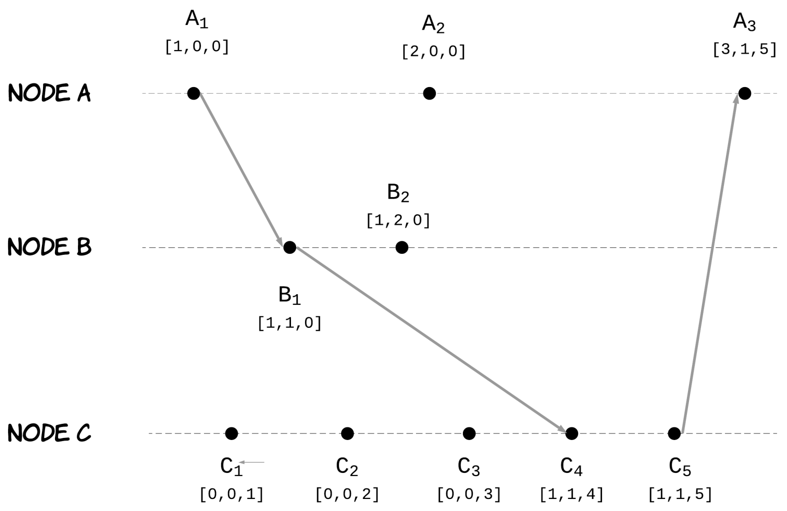


Figure 6.5: Vector clocks

that consists a source of concurrency needs to be considered as a node of the system in the context of the vector clocks, which means an entry for this process should be dedicated in each clock. For example, if our application consists of 3 servers and 2 clients, then each vector clock should contain 5 entries. Otherwise, we risk not being able to identify concurrent and conflicting operations, treating them as causally related instead[3]. It has been formally proven actually that the size of a vector clock must be at least `n` for a system consisting of `n` nodes in order to fully capture causality[52]. This means that vector clocks require a significant amount of storage in cases where the number of all the participating nodes is large, which can be the case for some types of systems nowadays, such as web applications where every browser can be considered a client of the system.

Version vectors & Dotted Version Vectors

There is a mechanism that is very similar to vector clocks, called **version vectors**. The data structure used by version vectors and the associated

[3]See: https://riak.com/posts/technical/why-vector-clocks-are-hard

update rules are very similar to those used by vector clocks. However, they are used for slightly different purposes. As explained previously, vector clocks are used to maintain a logical form of time, which can then be used to identify when events are happening, especially in comparison to other events. On the other hand, version vectors are better suited for applications that store data, where every data item is tagged with a version vector. In this way, data can potentially be updated in multiple parts of the system concurrently (e.g. when there is a network partition), so that the version vectors from the resulting data items can help us identify those items that can be reconciled automatically and those that require conflict resolution[53].

Version vectors maintain state identical to that in a vector clock, containing one integer entry per node in the system. The update rules are slightly different: nodes can experience both local updates (e.g. a write applied at a server) or can synchronize with another node (e.g. when recovering from a network partition).

- Initially, all vectors have all their elements set to zero.
- Each time a node experiences a local update event, it increments its own counter in the vector by one.
- Each time two nodes `a` and `b` synchronize, they both set the elements in their vector to the maximum of the elements across both vectors $V_a[x] = V_b[x] = max(V_a[x], V_b[x])$. After synchronisation, both nodes will have the same vectors. Furthermore, depending on whether the initial vectors were causally related or not, one of the associated items will supercede the other or some conflict resolution logic will be executed to maintain one entry associated with the new vector.

Version vectors are mostly beneficial in systems that act as datastores, so the nodes in the system will belong to two basic categories: the *server* or *replica nodes* that store the data and receive read/write operations and the *client nodes* that read data from the replica nodes and send update instructions. In many cases, clients might not even be part of our systems, such as scenarios where our system receives operations directly from web browsers of customers. As a result, it would be better to avoid significant amount of logic and storage overheads in the client nodes.

The version vector mechanism allows this in the following way: one entry is maintained for every node (both replica/server and client nodes). However, the client nodes can be stateless, which means they do not store the version vectors. Instead, they receive a version vector as part of every read operation and they provide this version vector when executing an update operation back to the corresponding replica node[4]. This vector is referred to as *context* and it's used by the replica node to update its version vector accordingly. Figure 6.6 contains an example execution in a distributed system using version vectors with one entry for each node in the system. For simplicity, the example assumes there is only one item, so all clients operate on the same item. It can easily be extended to cover cases with multiple items, where each item will have a separate version vector and the clients would also have to provide an identifier for the item to be accessed. Each read operation returns the current value with the corresponding version vector. Each write operation has 3 arguments: the first one is the version vector that is given as context, the second one is the identifier of the client node and the last one is the value to be written. The replica node is responsible for incrementing the appropriate entry in the vector (depending on the identifier of the client) and then persisting the new value. Note that this new value can either be persisted alongside other values that had been written concurrently or overwrite values that causally precede it. In the first case, multiple values will be returned in subsequent reads and the following write operations will reconcile them, persisting a single value. For instance, in our example below if node `D` performed a read from node `B` (which will return both values `V` and `W` and their version vectors) and then attempted to write the value `Z`, then the update will be of the form `PUT({(C,1), (D,1)}, D, Z)`. The replica node will calculate the new version vector `{(C,1), (D,2)}` and will identify it supersedes both existing version vectors `{(C,1)}` and `{(D,1)}`, so it will replace both values `V` and `W` with `Z`.

The approach of including entries in the vector clock for all client nodes is safe and can successfully identify when 2 different versions have been written concurrently or one of them causally precedes the other one and can be discarded. However, its main limitation is that the size of the vector clocks does not scale nicely. In distributed systems that are used as distributed datastores, the number of clients tends to be a lot bigger than the number of server nodes by two or three orders of magnitude. For instance, in many cases each item is replicated in 3 different servers, while there can be thousands of clients accessing this item. Note that even in cases where the clients of the systems are a few application servers, if each server is executing operations concurrently from multiple threads a separate entry in the vector clock needs

[4]This is only possible in an environment, where client nodes are not supposed to experience any local events, but only interact with server nodes via read/write operations. It also requires *read your writes* semantics (i.e. obtained via read/write quorums), so that each read returns the most recent update to a client[54].

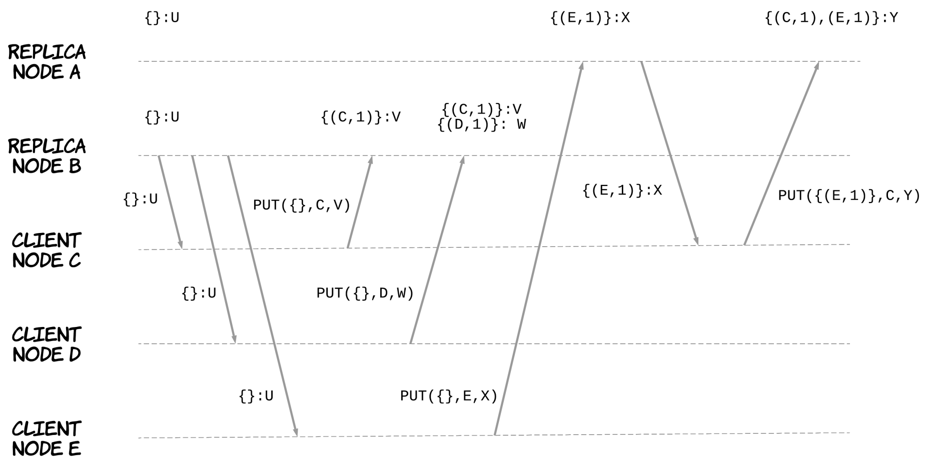


Figure 6.6: Version vectors with per-client entries

to be maintained for each one of them. As a result, this approach requires a significant amount of storage.

Ideally, we would like the size of the vector clocks to scale with the number of server nodes, instead of the number of clients. Could we perhaps remove the client entries from the vector clocks and let the servers increment their own entries, when performing the updates on behalf of the clients? Unfortunately, not. If we did this, the system would not be able to detect that some operations were performed concurrently, thus discarding values that should have been preserved. Figure 6.7 illustrates the issues with this approach. As you can see, the first write operations performed by client nodes C and D are concurrent. However, the server node B would not be able to identify that. The node B would consider the version vector of the second update `{(B,2)}` to supercede `{(B,1)}`, thus discarding the value V and replacing it with the value W.

There is a technique that makes it possible to successfully identify concurrent versions, while also allowing the version vectors to scale with the number of servers, called **dotted version vectors**. This technique has a characteristic that allows it to achieve this: each entry in the vector is not anymore a single number, but a pair of numbers. This can encode a sequence of numbers that are not fully sequential, but can contain one gap. The pair (n_1,n_2) represents all the numbers from 1 to n_1 plus the number n_2. For example,

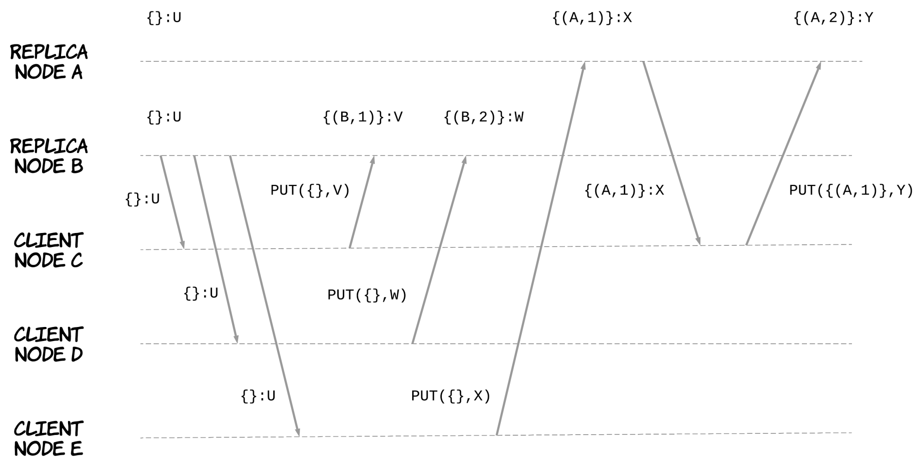


Figure 6.7: Version vectors with per-server entries

the pair `(4,7)` represents the sequence `[1,2,3,4,7]`. Note that the second number is optional and some entries can still be single number. This can be leveraged to keep track of concurrency between multiple versions. The order between 2 versions is now defined in terms of the *contains* relationship on the corresponding sequences. So, for vectors v_1, v_2 the relationship $v_1 \leq v_2$ holds if the sequence represented by v_1 is a subset of the sequence represented by v_2. More specifically:

- $(m) \leq (m')$ if $m \leq m'$
- $(m) \leq (m', n')$ if $m \leq m' \lor m = m' + 1 = n'$
- $(m, n) \leq (m')$ if $n \leq m'$
- $(m, n) \leq (m', n')$ if $n \leq m' \lor (m \leq m' \land n = n')$

The update rule executed by each replica node when receiving a write operation is also slightly different. For all the indexes except the one belonging to the replica node, the node uses the value `(m)` where `m` is the maximum number amongst those available in the provided version vectors in the context. For the index corresponding to the replica node, the node uses the pair `(m, n+1)`, where `m` is the maximum number amongst those available in the provided version vectors in the context and `n` is the maximum number amongst the version vectors present in the replica node (essentially the value of its logical clock). For a more elaborate analysis of the rules and a formal proof that this technique is safe, refer to the original paper[54]. Figure 6.8

illustrates what a solution with dotted version vectors would look like in our previous examples. As you can see, the write operations by client nodes C and D end up receiving version vectors `{(B,0,1)}` and `{(B,0,2)}` and they are successfully identified as concurrent, since `{(B,0,1)}` $\not\leq$ `{(B,0,2)}` and `{(B,0,2)}` $\not\leq$ `{(B,0,1)}`.

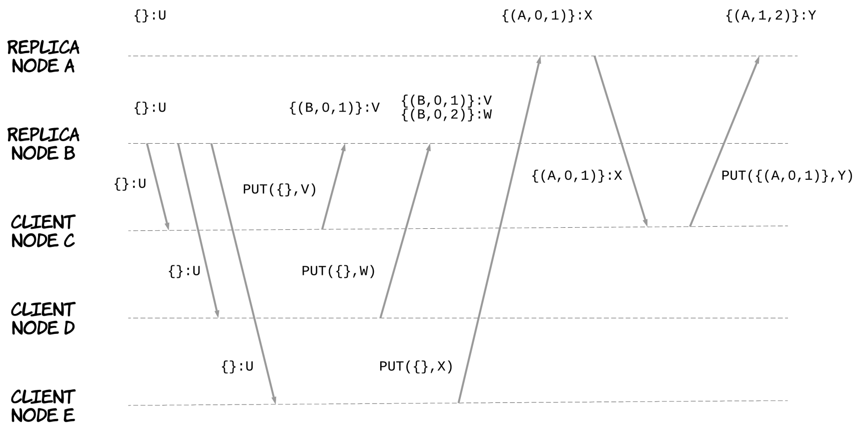


Figure 6.8: Dotted version vectors

Distributed snapshots

There is another basic problem in the field of distributed systems that is strongly related to the notion of time and order: how to record a **snapshot** of the state of a distributed system composed of multiple nodes that perform a continuous computation. This snapshot can simply be used as a recovery mechanism from a point in the past when failures happen. However, there are many more problems in distributed systems that can be expressed in terms of the problem of detecting a global state and specific properties associated with it[5]. This section will cover a seminal algorithm used for capturing distributed snapshots, known as the Chandy-Lamport algorithm named after its inventors [55].

The state of a distributed system consists of the state of the various nodes

[5]This problem is also known as *stable property detection* and can have many different usages, such as detection of deadlocks or termination of a computation.

and any messages that are in transit between the nodes. The main challenge in recording this state is that the nodes that are part of the system do not have a common clock, so they cannot record their local states at precisely the same instant. As a result, the nodes have to coordinate with each other by exchanging messages, so that each node records its state and the state of associated communication channels. Thus, the collective set of all node and channel states will form a global state. Furthermore, any communication required by the snapshot protocol should not alter the underlying computation.

The paper presents a very interesting and illuminating analogy for this problem. Imagine a group of photographers observing a panoramic, dynamic scene such as a sky filled with migrating birds. This scene is so big that it cannot be captured by a single photograph. As a result, the photographers must take several snapshots and piece them together to form a picture of the overall scene. The snapshots cannot be taken at the same time and the photographers should not disturb the process that is being photographed, i.e. they cannot get all the birds to remain motionless while the photographs are taken. However, the composite picture should be *meaningful*.

This need for a meaningful snapshot still exists when talking about distributed systems. For example, there's no point recovering from a snapshot, if that snapshot can lead to the system to an erroneous or corrupted state. A meaningful snapshot is termed as a **consistent** snapshot in the paper, which presents a formal definition of what this is[6]. This definition will be presented here in a more simplified way for ease of understanding. Let's assume a distributed system can be modelled as a directed graph, where vertices represent nodes of the system and edges represent communication channels. An event `e` in a node `p` is an atomic action that may change the state of `p` itself and the state of at most one channel `c` incident on p: the state of `c` may be changed by the sending of a message `M` along `c` (if `c` is an outbound edge from `p`) or the receipt of a message `M` along `c` (if `c` is an inbound edge to `p`). So, an event `e` could be represented by the tuple `<p, s, s', M, c>`, where `s` and `s'` are the previous and new state of the node. An event e_i moves the global state of the system from S_i to S_{i+1}. A snapshot $S_{snapshot}$ is thus consistent if:

[6]An alternative definition is that of a *consistent cut* [51], which partitions the space-time diagram along the time axis in a way that respects causality, e.g. for each pair of events `e` and `f`, if `f` is in the cut and `e -> f` then `e` is also in the cut. Note that the Chandy-Lamport algorithm produces snapshots that are also consistent cuts.

- $\mathtt{S}_{\text{snapshot}}$ is reachable from the state $\mathtt{S}_{\text{start}}$ in which the algorithm was initiated.
- the state $\mathtt{S}_{\text{end}}$ in which the algorithm terminates is also reachable from $\mathtt{S}_{\text{snapshot}}$.

Let's look at an example to get some intuition about this. Let's assume we have a very simple distributed system consisting of 2 nodes `p`, `q` and two channels `c`, `c'`, as shown in Figure 6.9. The system contains one token that is passed between the nodes. Each node has two possible states $\mathtt{s}_0$ and $\mathtt{s}_1$, where $\mathtt{s}_0$ is the state in which the node does not possess the token and $\mathtt{s}_1$ is the state in which it does. Figure 6.10 contains the possible global states of the systems and the associated transitions. As a result, a snapshot where the state is $\mathtt{s}_0$ for both of the nodes and the state of both channels is empty would not be consistent, since the token is lost! A snapshot where the states are $\mathtt{s}_1$ and $\mathtt{s}_0$ and channel `c` contains the token is also not consistent, since there are now two tokens in the system.

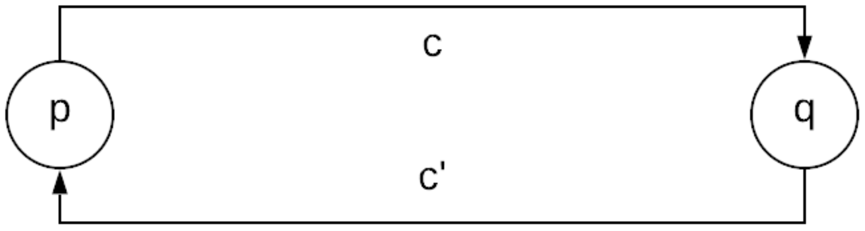


Figure 6.9: A simple distributed system consisting of 2 nodes and 2 communication channels

The algorithm is based on the following main idea: a *marker* message is sent between nodes using the available communication channels that represents an instruction to a node to record a snapshot of the current state. The algorithm works as follows:

- The node that initiates the protocol records its state and then sends a marker message to all the outbound channels. Importantly, the marker is sent after the node records its state and before any further messages are sent to the channels.
- When a node receives a marker message, the behaviour depends on whether the node has already recorded its state (while emitting the

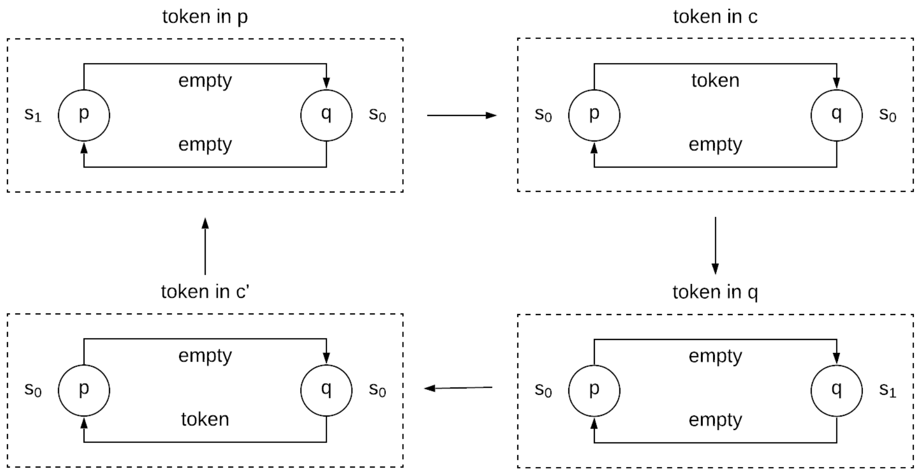


Figure 6.10: The possible global states and the corresponding transitions of the token system

mark previously) or not.

- If the node has not recorded its state, it records its state and then it records the state of the channel `c` the marker was received from as an empty sequence. It then sends the marker to all the outbound channels.
- If the node has recorded its state, it records the state of the channel the marker was received from as the sequence of messages received from `c` after the node's state was recorded and before the node received the marker from `c`.

Figure 6.11 contains a sample execution of this algorithm on the simple system presented previously. The node `p` sends the token and right after initiates an execution of the protocol. As a result, it records its state s_0 and sends the marker in channel `c`. The node `q` receives the token, transitions to state s_1. It then sends the token to channel `c'` and transitions to state s_0. Afterwards, it receives the marker message, records its state s_0 and the state of the channel `c` as an empty sequence and sends the marker message to channel `c'`[7]. In the meanwhile, node `p` has received the token, transitioned

[7]Note that this is just one of the possible executions. In an alternative execution, the node `q` could have processed both the token and the marker, recording its state as s_1 and potentially sending the marker across channel `c'` without sending the token yet. This would have led to a different, but still consistent snapshot.

to state `s1` and buffered the token in the sequence of messages received while the snapshot protocol was executing. The node `p` then receives the marker and records the state of the channel `c'` as the sequence `[token]`. At this point, the protocol concludes, since the state of all nodes and channels has been recorded and the global snapshot state is the following:

```
snapshot(p): s0
snapshot(q): s0
snapshot(c): []
snapshot(c'): [token]
```

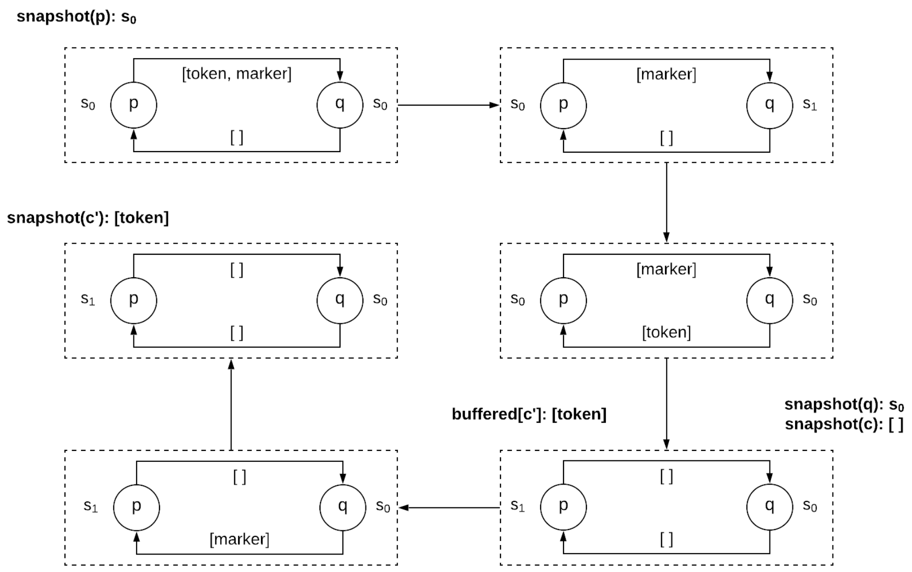


Figure 6.11: An execution of Chandy-Lamport algorithm on the token system

Physical & Logical time: closing thoughts

Hopefully, the two previous chapters helped you understand the difference between the concepts of physical and logical time. At the same time, some parts went into detail to explain the inner workings of some techniques and their benefits and pitfalls. This might have left you with more questions, so this section will contain an overview of what we have seen and some additional observations. The goal is to help you finish this chapter with a clear understanding of the difference between these 2 concepts and what each one has to offer.

Let's start by reviewing the basic difference between *physical* and *logical time*. As explained previously, physical time is measured based on some physical phenomena. Depending on the phenomenon that is observed, the granularity of physical time can differ from days to seconds or nanoseconds. In all cases, the time *flows* continually between discrete values[8]. In a non-distributed system, a computing node can use these measurements to associate occuring events and compare them with each other to determine which happened first. In a distributed system, the various nodes have separate clocks that might not be in sync, so they will have to do additional work to make sure the values from their clocks can be compared safely. This additional work will involve exchange of messages between nodes, which will contain the values of their clocks, so that nodes can adjust their clocks accordingly to synchronize them. On the other hand, logical time is not measured based on physical phenomena, but the node makes use of local events to measure logical time instead. For instance, the node can maintain a counter that is incremented everytime something happens. In this way, every event can still be associated with a discrete "moment in time", but the value of this time will be mostly relative and not absolute. Furthermore, logical time will not flow continuously, but it will only flow when events happen at a node. The node will still be able to compare local events and determine which one happened first. As mentioned before, these instances of time do not have any absolute semantics. As a result, in a distributed system, the various nodes will have to exchange their perception of logical time, so that they can compare the values of their logical clocks and be able to order events across different nodes (and clocks).

After reading the previous paragraph, did you notice anything interesting? In the beginning, the concepts of physical and logical time do seem really different. However, closer examination reveals they are quite similar and actually share some basic properties. In both cases, time flows in discrete increments everytime something happens. In the case of physical time, what happens is the underlying physical phenomenon, while in the case of logical time it's the actual logical event that happens in a node. In both cases, communication is required between nodes to synchronise their

[8]Time can be considered discrete in the context of hardware or software systems. Whether this is true in other contexts (i.e. physics) is a much bigger question and is out of topic for this discussion.

clocks. In the case of physical time, this communication is performed in the background continuously, while in the case of logical time, this communication is performed *on-demand*, when messages are sent between nodes.

Identifying that these 2 notions make use of these common, basic principles but in a slightly different way is useful to understand the advantages and disadvantages of each concept and where each one can be useful. By its nature, physical time attempts to perform all the necessary coordination and communication in the background, so that the various nodes can establish an order without additional coordination, when needed. In most cases, physical clocks are used in order to provide the illusion of a total order, which is not realistic in a distributed system, as explained before. Furthermore, physical clocks need to remain properly synchronised, which might not be temporarily possible under network partitions. If clock drift between nodes is larger than the acceptable error, then this might mean the correctness of the application is compromised. On the other hand, logical time is operating under the assumption that network partitions are a given and a partial order satisfying causality is the best kind of ordering one can achieve in a distributed system. During partitions, different nodes can keep operating, while essentially leaving their clocks *drift*. However, when partitions are healed or nodes communicate with each other, they can eventually detect things that happened concurrently and perform the necessary reconciliation. Logical clocks can also be adapted, so that only necessary causal relationships are captured (instead of everything), thus achieving both good performance and safety. However, as explained previously logical time does not flow on its own as physical time does, it only flows when events happen. As a consequence, logical time cannot be used for tasks that require a notion of wall clock time that flows even when no events happen. For instance, in order to identify whether another node is slow during communication or has potentially crashed, a node needs to make use of timeouts and physical time.

Part IV

From theory to practice

Chapter 7

Case studies

This chapter examines in more detail some specific examples of distributed systems. Some of these systems are commercially available and widely used. Some of them have been developed and used internally in companies, but their design has been shared publicly via academic papers. The goal of this chapter is twofold: first to provide an overview of the basic categories of distributed systems and then to explain how these systems make use of the principles that have been described so far.

However, keep in mind that some of the systems described in this chapter might have evolved since the time of writing. As a point of reference, the table below contains the versions of the associated systems at the time of writing, when applicable.

System	Version
HDFS	3.1.2
Zookeeper	3.5.5
Hbase	2.0
Cassandra	3.11.4
FaunaDB	2.7
Kafka	2.3.1
Kubernetes	1.13.12
Corda	4.1
Spark	2.4.4
Flink	1.8

Distributed file systems (HDFS/GFS)

Google File System (GFS)[56] is a proprietary distributed file system developed by Google, which has been the inspiration for the Hadoop Distributed File System (HDFS)[57], a distributed file system that has been developed as an Apache project[1]. As explained later, the basic design principles are similar for these two systems, which also have some small differences.

The core requirements for these distributed file systems were the following:

- **fault tolerance**: the ability to be operational even when some machines of the system have failed.
- **scalability**: the ability to scale the system to significantly large sizes both in terms of number of files and file sizes.
- **optimised for batch operations**: the system should be optimised for use-cases that involved batch operations, such as applications that performed processing and analysis of huge datasets. This implies that throughput is more important than latency and most of the files are expected to be mutated by appending data rather than overwriting existing data.

Figure 7.1 displays a high-level overview of the GFS architecture. A GFS cluster consists of a **single master node** and **multiple chunkserver nodes**. Chunkserver nodes are responsible for storing and serving the data of the files, while the master node is responsible for maintaining the file system metadata, informing clients about which chunkservers store a specific part of a file and performing necessary administration tasks, such as garbage collection of orphaned chunks or data migration during failures. Note that the HDFS architecture is similar, but the *master node* is called *Namenode* and the *chunkserver nodes* are called *Datanodes*.

Each file is divided into fixed-size *chunks*, which are identified by an immutable and globally unique 64-bit *chunk handle*, assigned by the master during chunk creation. Chunkservers store chunks on local disks as regular files. The system employs both **partitioning** and **replication**: it partitions files across different chunkservers and replicates each chunk on multiple chunkservers. The former improves **performance** and the latter improves **availability** and **data reliability**. The system takes into account the network topology of a datacenter, which usually consists of multiple racks of servers. This has several implications, e.g. bandwidth into or out of a rack

[1]See: https://www.apache.org

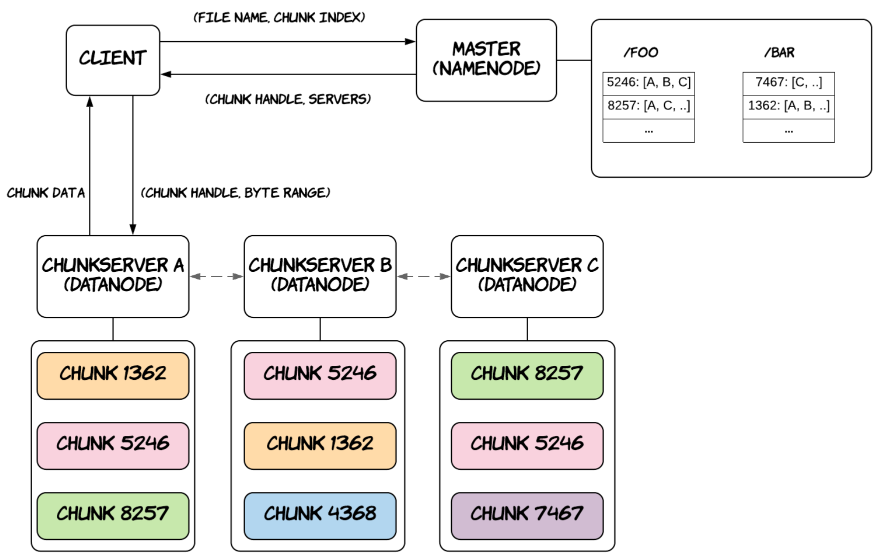


Figure 7.1: GFS high-level architecture

may be less than the aggregate bandwidth of all the machines within the rack and a failure of a single shared resource in a rack (a network switch or power circuit) can essentially bring all the machines of the rack down. When creating a new chunk and placing its initially empty replicas, a master tries to use chunkservers with below-average disk space utilisation. It also tries to use chunkservers that have a low number of recent creations, since that can reliably predict imminent heavy write traffic. In this way, the master attempts to balance disk and network bandwidth utilisation across the cluster. When deciding where to place the replicas, the master also follows a chunk replica placement policy that is configurable. By default, it will attempt to store two replicas at two different nodes that reside in the same rack, while storing the third replica at a node that resides in a separate rack. This is a trade-off between high network bandwidth and data reliability.

The clients can create, read, write and delete files from the distributed file system by using a GFS client library linked in to the application that abstracts some implementation details. For example, the applications can operate based on byte offsets of files and the client library can translate these byte offsets to the associated chunk index, communicate with the master to retrieve the chunk handle for the provided chunk index and the location of the associated chunkservers and finally contact the appropriate chunkserver (most likely the closest one) to retrieve the data. Figure 7.1 displays this workflow for a read operation. Clients cache the metadata for chunk locations locally, so they only have to contact master for new chunks or when the cache has expired. During migration of chunks due to failures, clients organically request fresh data from the master, when they realise the old chunkservers cannot serve the data for the specified chunk anymore. On the other hand, clients do not cache the actual chunk data, since they are expected to stream through huge files and have working sets that are too large to benefit from caching.

The master stores the file and chunk namespaces, the mapping from files to chunks and the chunk locations. All metadata is stored in the master's memory. The namespaces and the mappings are also kept persistent by logging mutating operations (e.g. file creation, renaming etc.) to an operation log that is stored on the master's local disk and replicated on remote machines. The master node also checkpoints its memory state to disk when the log grows significantly. As a result, in case of the master's failure the image of the filesystem can be reconstructed by loading the last checkpoint in memory and replaying the operation log from this point forward. File namespace mutations are atomic and linearizable. This is achieved by executing this operation in a single node, the master node. The operation log defines a global total order for these operations and the master node also makes use of read-write locks on the associated namespace nodes to perform proper serialization on any concurrent writes.

GFS supports multiple concurrent writers for a single file. Figure 7.2 illustrates how this works. The client first communicates with the master node to identify the chunkservers that contain the relevant chunks. Afterwards, the clients starts pushing the data to all the replicas using some form of *chain replication*. The chunkservers are put in a chain depending on the network topology and data is pushed linearly along the chain. For instance, the client pushes the data to the first chunkserver in the chain, which pushes the data to the second chunkserver etc. This helps fully utilize each machine's network bandwidth avoiding bottlenecks in a single node. The master grants a lease for each chunk to one of the chunkservers, which is nominated as the *primary replica*, which is responsible for serializing all the mutations on this chunk. After all the data is pushed to the chunkservers, the client sends a write request to the primary replica, which identifies the data pushed earlier. The primary assigns consecutive serial numbers to all the mutations, applies them locally and then forwards the write request to all secondary replicas, which apply the mutations in the same serial number imposed by the primary. After the secondary replicas have acknowledged the write to the primary replica, then the primary replica can acknowledge the write to the client.

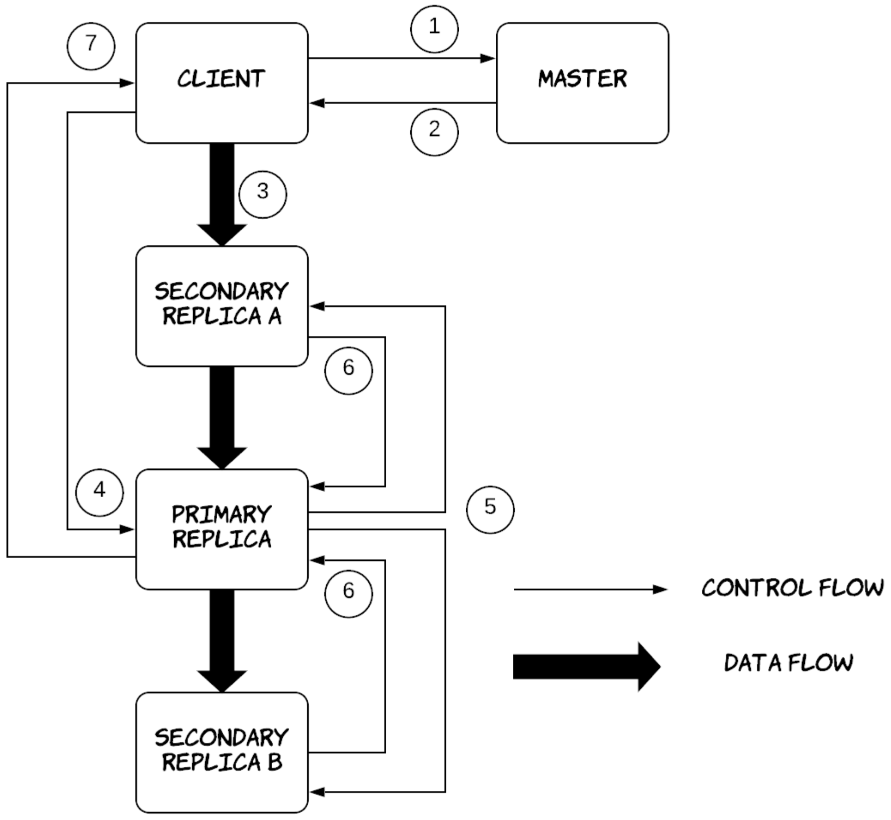
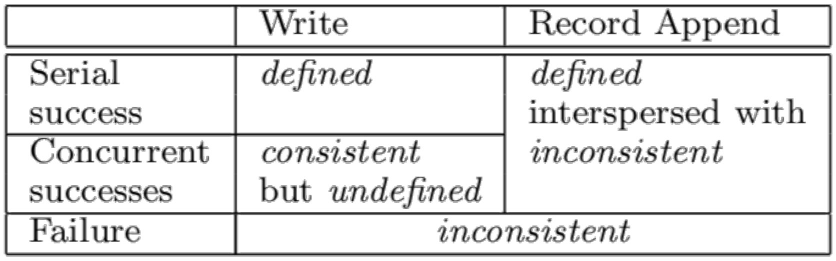


Figure 7.2: How writes work in GFS

Of course, this flow is vulnerable to partial failures. For example, think about the scenario, where the primary replica crashes in the middle of performing a write. After the lease expires, a secondary replica can request the lease and start imposing a new serial number that might disagree with the writes of other replicas in the past. As a result, a write might be persisted only in some replicas or it might be persisted in different orders in different replicas. GFS provides a **custom consistency model** for write operations. The state of a file region after a mutation depends on the type of mutation, whether it succeeds or fails and whether there are concurrent mutations. A file region is **consistent** if all clients will always see the same data, regardless of the replica they read from. A region is *defined* after a file data mutation if it is consistent and clients will see what the mutation writes in its entirety. When a mutation succeeds without interference from concurrent writes, the affected region is defined: all clients will always see what the mutation has written. Concurrent successful mutations leave the region undefined but consistent: all the clients see the same data, but it may not reflect what any one mutation has written. Typically, it consists of mingled fragments from multiple mutations. A failed mutation makes the region inconsistent: different clients may see different data at times. Besides regular writes, GFS also provides an extra mutation operation: **record appends**. A record append causes data to be appended *atomically at least once* even in the presence of concurrent mutations, but at an offset of GFS's choosing, which is returned to the client. Clients are supposed to retry failed record appends and GFS guarantees that each replica will contain the data of the operation as an atomic unit at least once in the same offset. However, GFS may insert padding or record duplicates in between. As a result, successful record appends create defined regions interspersed with inconsistent regions. Table 7.3 contains a summary of the GFS consistency model.



	Write	Record Append
Serial success	*defined*	*defined* interspersed with *inconsistent*
Concurrent successes	*consistent* but *undefined*	
Failure	*inconsistent*	

Figure 7.3: How writes work in GFS

Applications can accommodate this relaxed consistency model of GFS by applying a few simple techniques at the application layer: using appends rather than overwrites, checkpointing and writing self-validating, self-identifying records. Appending is far more efficient and more resilient to application failures than random writes. Each record prepared by a writer can contain extra information like checksums so that its validity can be verified. A reader can then identify and discard extra padding and record fragments using these checksums. If occasional duplicates are not acceptable, e.g. if they could trigger non-idempotent operations, the reader can filter them out using unique record identifiers that are selected and persisted by the writer.

HDFS has taken a slightly different path to simplify the semantics of mutating operations. Specifically, HDFS supports only a single writer at a time. It provides support only for append (and not overwrite) operations. It also does not provide a record append operation, since there are no concurrent writes and it handles partial failures in the replication pipeline a bit differently, removing failed nodes from the replica set completely in order to ensure file content is the same in all replicas.

Both GFS and HDFS provide applications with the information where a region of a file is stored. This enables the applications to schedule processing jobs to be run in nodes that store the associated data, minimizing network congestion and improving the overall throughput of the system. This principle is also known as **moving computation to the data**.

Distributed coordination service (Zookeeper/Chubby/etcd)

It must have become evident by now that **coordination** is a central aspect in distributed systems. Even though each component of a distributed system might function correctly in isolation, one needs to ensure that they will also function correctly when operating simultaneously. This can be achieved through some form of coordination between these components. As illustrated in the section about consensus, this coordination can end up being quite complicated with many edge cases. As a consequence, implementing these coordination algorithms on every new system from scratch would be inefficient and would also introduce a lot of risk for bugs. On the contrary, if there was a separate system that could provide this form of coordination as an API, it would be a lot easier for other systems to offload any coordination function to this system.

Several different systems were born out of this need. Chubby [58] was such a system implemented internally in Google and used from several different systems for coordination purposes. Zookeeper [59] was a system that was partially inspired from Chubby, it was originally developed in Yahoo and later became an Apache project. It has been widely used by many companies to perform coordination in distributed systems, including some systems that are part of the Hadoop ecosystem. etcd[2] is another system that implements similar coordination primitives and formed the basis of Kubernetes' control plane. As expected, these systems present a lot of similarities, but they also have some small differences. For the sake of brevity, this chapter will focus on Zookeeper providing an overview of its design, but it will also try to comment on the basic differences of the other two systems where relevant.

Let's start by looking at Zookeeper's API, which is essentially a hierarchical namespace similar to a filesystem.[3] Every name is a sequence of path elements separated by a slash (`/`). Every name represents a data node (called *znode*), which can contain a piece of metadata and children nodes. For instance, the node `/a/b` is considered a child of the node `/a`. The API contains basic operations that can be used to create nodes, delete nodes, check if a specific node exists, list the children of a node and read or set the data of a node. There are 2 types of znodes: *regular nodes* and *ephemeral nodes*. Regular nodes are created and deleted explicitly by clients. Ephemeral nodes can also be removed by the system when the session that created them expires (i.e. due to a failure). Additionally, when a client creates a new node, it can set a *sequential flag*. Nodes created with this flag have the value of a monotonically increasing counter appended to a provided prefix. Zookeeper also provides an API that allows clients to receive notifications for changes without polling, called *watches*. On read operations, clients can set a watch flag, so that they are notified by the system when the information returned has changed. A client connects to Zookeper initiating a session, which needs to be maintained open by sending heartbeats to the associated server. If a Zookeeper server does not receive anything from a client for more than a specified timeout, it considers the client faulty and terminates the session. This deletes the associated ephemeral nodes and unregisters any watches registered via this session. The update operations can take an expected version number, which enables the implementation of conditional updates resolving any conflicts arising from concurrent update requests.

Zookeeper nodes form a cluster, which is called a *Zookeepe ensemble*. One of these nodes is designated as the **leader** and the rest of the nodes are **followers**. Zookeeper makes use of a custom atomic broadcast protocol, called Zab [60][61]. This protocol is used in order to elect the leader and

[2]See: https://etcd.io

[3]Chubby also provides a hierarchical namespace, while etcd provides a key-value interface.

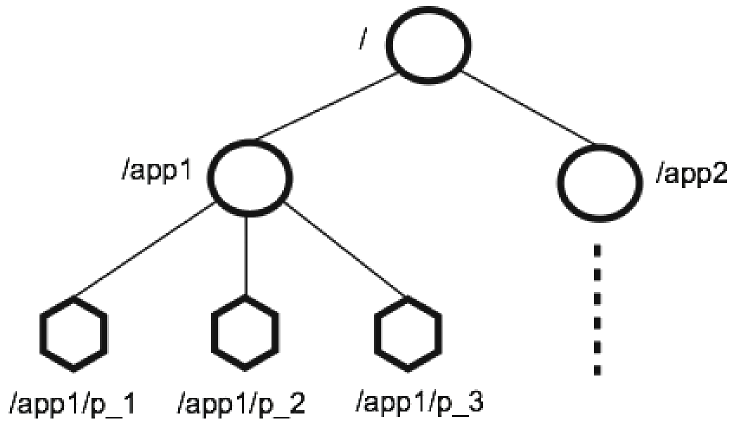


Figure 7.4: Hierarchical namespace in Zookeeper

replicate the write operations to the followers[4]. Each of those nodes has a copy of the Zookeper state in memory. Any changes are also recorded in a durable, write-ahead log which can be used for recovery. All the nodes can serve read requests using their local database. Followers have to forward any write requests to the leader node, wait until the request has been successfully replicated/broadcasted and then respond to the client. Reads can be served locally without any communication between nodes, so they are extremely fast. However, a follower node might be lagging behind the leader node, so client reads might not necessarily reflect the latest write that has been performed. For this reason, Zookeeper provides an additional operation called `sync`. Clients can initiate a `sync` before performing a read. In this way, the read will reflect any write operations that had happened before the `sync` was issued. The `sync` operation does not need to go through the broadcast protocol, it it just placed at the end of the leader's queue and forwarded only to the associated follower[5].

As a result, Zookeeper provides the following 2 safety guarantees:

- **Linearizable writes**: all requests that update the state of Zookeeper are serializable and respect precedence. As mentioned before, writes

[4]Chubby uses Paxos for this purpose, while etcd makes use of Raft.

[5]In contrast, in Chubby both read and write requests are directed to the master. This has the benefit of increased consistency, but the downside of decreased throughput. To mitigate this, Chubby clients cache extensively and the master is responsible for invalidating the caches before completing writes, thus making the system a bit more sensitive to client failures.

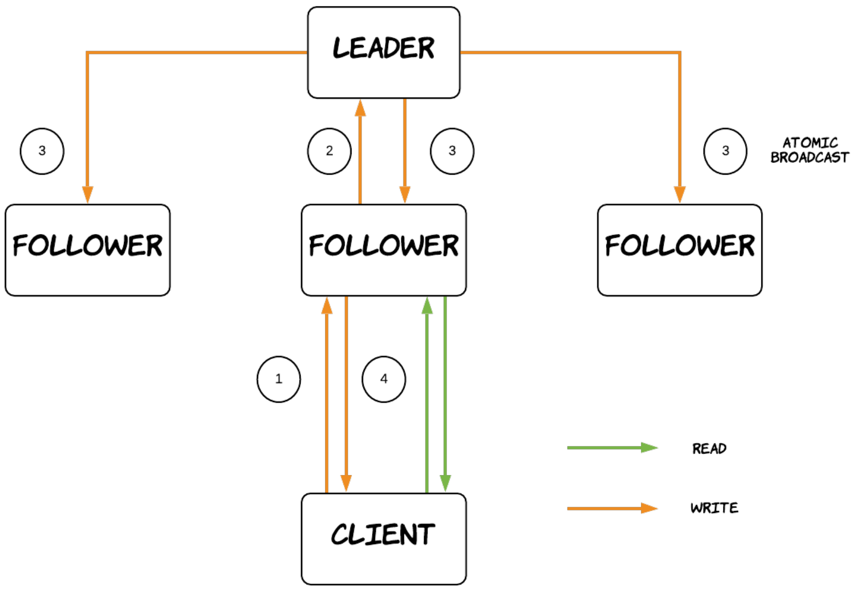


Figure 7.5: Zookeeper architecture

are not linearizable with respect to reads, if `sync` is not used.

- **FIFO client order**: all requests from a given client are executed in the order they were sent by the client.

There is one more important ordering guarantee: if a client is waiting for a change, the client will see the notification event before it sees the new state of the system after the change is made. As a result, when a client receives a notification and performs a read, the result will reflect all writes at least up to the one that triggered this notification.

Zookeeper also provides the following 2 liveness and durability guarantees:

- if a majority of servers are active and communicating, the service will be available
- if the service responds successfully to a change request, that change persists across any number of failures as long as a quorum of servers is eventually able to recover.

As mentioned above, the Zookeeper atomic broadcast protocol (ZAB) is used in order to agree on a leader in the ensemble, synchronize the replicas, manage the broadcast of update transactions and recover from a crashed state to a valid state. This protocol shares a lot of characteristics with other consensus protocols, such as Paxos or Raft. In Zab, transactions are identified by a specific type of identifier, called **zxid**. This identifier consists of 2 parts `<e, c>`, where `e` is the epoch number of the leader that generated the transaction and `c` is an integer acting as a counter for this epoch. The counter `c` is incremented every time a new transaction is introduced by the leader, while `e` is incremented when a new leader becomes active. The protocol consists of 4 basic phases:

- **Leader election**: peers are initialised in this phase, having state election. This phase terminates when a quorum of peers have voted for a leader. This leader is *prospective* and will become an *established leader* only after the end of phase 3.
- **Discovery**: in this phase, the leader communicates with the followers in order to discover the most up-to-date sequence of accepted transactions among a quorum and establish a new epoch so that previous leaders cannot commit new proposals.
- **Synchronisation**: in this phase, the leader synchronises the replicas in the ensemble using the leader's updated history from the previous phase. At the end of this phase, the leader is said to be *established*.
- **Broadcast**: in this phase, the leader receives write requests and performs broadcasts of the associated transactions. This phase lasts until the leader loses its leadership, which is essentially maintained via heartbeats to the followers.

In practice, Zookeeper is using a leader election algorithm called Fast Leader Election (FLE), which employs an optimisation. It attempts to elect as leader the peer that has the most up-to-date history from a quorum of processes. This is done in order to minimize the data exchange between the leader and the followers in the Discovery phase.

The Zookeeper API can be used to build more powerful primitives. Some examples are the following:

- **Configuration management**: This can be achieved simply by having the node that needs to publish some configuration information create a znode z_c and write the configuration as the znode's data. The znode's path is provided to the other nodes of the system, which obtain the configuration by reading z_c. They also register a watch, so that they are informed when this configuration changes. If that happens, they are notified and perform a new read to get the latest configuration.

- **Group membership**: A node z_g is designated to represent the group. When a node wants to join the group, it creates an ephemeral child node under z_g. If each node has a unique name or identifier, this can be used as the name of the child node. Alternatively, the nodes can make use of the sequential flag to obtain a unique name assignment from Zookeeper. These nodes can also contain additional metadata for the members of the group, such as addresses and ports. Nodes can obtain the members of the group by listing the children of z_g. If a node wants to also monitor changes in the group membership, it can register a watch. When nodes fail, their associated ephemeral nodes are automatically removed, which signals their removal from the group.
- **Simple locks**: The simplest way to implement locks is by using a simple "*lock file*", which is represented by a znode. To acquire a lock, a client tries to create the designated znode with the ephemeral flag. If the create succeeds, the client holds the lock. Otherwise, the client can set a watch to the created node to be notified if the lock is released, so that it attempts to re-acquire it. The lock is released when the client explicitly deletes the znode or when it dies.
- **Locks without herd effect**: The previous pattern suffers from the herd effect: if there are many clients waiting to acquire a lock, they will all be notified simultaneously and attempt to acquire the lock even though only one can acquire it, thus creating unnecessary contention. There is a different way to implement locks to avoid this problem. All the clients competing for the lock attempt to create a sequential, ephemeral znode with the same prefix (i.e. `/lock-`). The client with the smallest sequence number acquires the lock. The rest of the clients register watches for the znode with the next lower sequence number. Once a node is notified, it can check if it's now the lowest sequence number, which means it has acquired the lock. Otherwise, it registers a new watch for the next znode with the lower sequence number.

In similar ways, many more primitives can be built, such as read/write locks, barriers etc. These patterns in Zookeeper are usually called *recipes*[6].

[6]See: https://zookeeper.apache.org/doc/current/recipes.html

Distributed datastores

This section will examine some basic categories of distributed datastores. It is impossible to cover all available datastores here. Furthermore, each datastore can have some special characteristics that make it differ from the rest, so it is not easy to establish clear boundaries between them and classify them into well-defined categories. For this reason, datastores are grouped based on their most basic architectural characteristics and their historical origins. It is useful to draw comparisons against these systems and try to understand the strengths and weaknesses of each one.

BigTable/HBase

BigTable [7] is a distributed storage system that was initially developed in Google and was the inspiration for HBase[7], a distributed datastore that is part of the Apache Hadoop project. As expected, the architecture of these two systems is very close, so this section will focus on HBase, which is an open-source system.

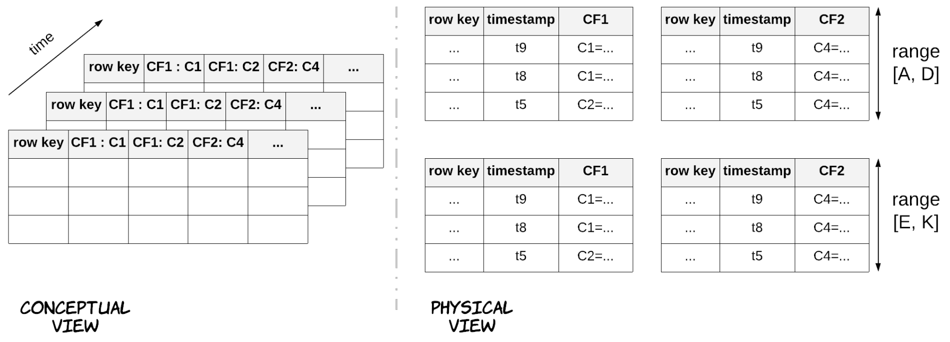


Figure 7.6: HBase data model & physical layout

HBase provides a sparse, multi-dimensional sorted map as a data model, as shown in Figure 7.6. The map is indexed by a row key, a column key and a timestamp, while each value in the map is an uninterpreted array of bytes. The columns are further grouped in column families. All members of a column family are physically stored together on the filesystem and

[7]See: https://hbase.apache.org

the user can specify tuning configurations for each column family, such as compression type or in-memory caching. Column families need to be declared upfront during schema definition, but columns can be created dynamically. Furthermore, the system supports a small number of column families, but an unlimited number of columns. The keys are also uninterpreted bytes and rows of the table are physically stored in lexicographical order of the keys. Each table is **partitioned horizontally** using **range partitioning** based on the row key into segments, called regions. The main goal of this data model and the architecture described later is to allow the user to control the physical layout of data, so that related data are stored near each other.

Figure 7.7 shows the high-level architecture of HBase, which is also based on a master-slave architecture. The master is called **HMaster** and the slaves are called **region servers**. The HMaster is responsible for assigning regions to region servers, detecting the addition and expiration of region servers, balancing region server load and handling schema changes. Each region server manages a set of regions, handling read and write requests to the regions it has loaded and splitting regions that have grown too large. Similar to other single-master distributed systems, clients do not communicate with the master for data flow operations, but only for control flow operations in order to prevent it becoming the performance bottleneck of the system. Hbase uses Zookeeper to perform **leader election** of the master node, maintain **group membership** of region servers, store the bootstrap location of HBase data and also store schema information and access control lists. Each region server stores the data for the associated regions in HDFS, which provides the necessary redundancy. A region server can be collocated at the same machine of an HDFS datanode to enable data locality and minimize network traffic.

There is a special HBase table, called the `META` table, which contains the mapping between regions and region servers in the cluster. The location of this table is stored in Zookeeper. As a result, the first time a client needs to read/write to HBase, it first communicates with Zookeeper to retrieve the region server that hosts the `META` table, then contacts this region server to find the region server that contains the desired table and finally sends the read/write operation to that server. The client caches locally the location of the `META` table and the data already read from this table for future use. HMasters initiallly compete to create an ephemeral node in Zookeeper. The first one to do so becomes the active master, while the second one listens for notifications from Zookeeper of the active master failure. Similarly, region servers create ephemeral nodes in Zookeeper at a directory monitored by the

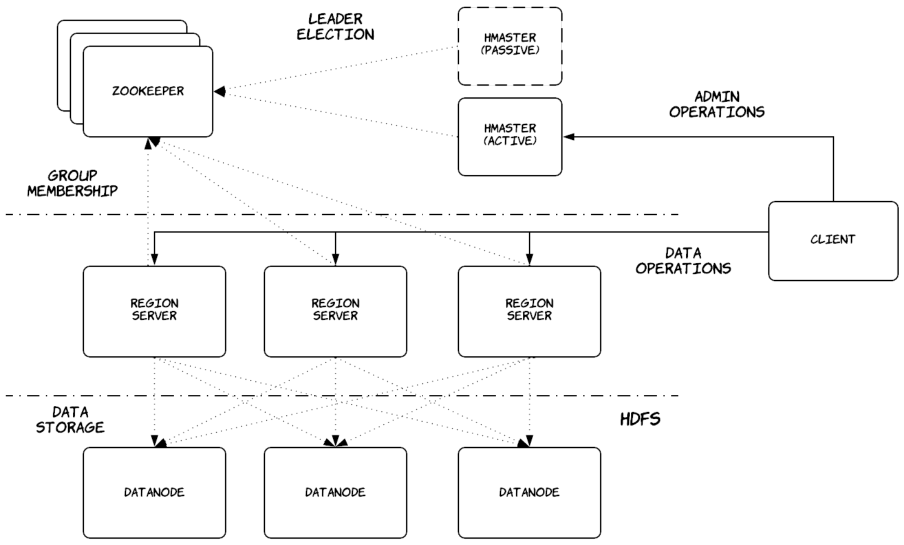


Figure 7.7: HBase high-level architecture

HMaster. In this way, the HMaster is aware of region servers that join/leave the cluster, so that it can manage assignment of regions accordingly.

Appends are more efficient than random writes, especially in a filesystem like HDFS. Region servers try to take advantage of this fact by employing the following components for storage and data retrieval:

- **MemStore**: this is used as a *write cache*. Writes are initially written in this data structure, which is stored in-memory and can be sorted efficiently before being written to disk. Writes are buffered in this data structure and periodically written to HDFS after being sorted.
- **HFile**: this is the file in HDFS which stores sorted key-value entries on disk.
- **Write ahead log (WAL)**: this stores operations that have not been persisted to permanent storage and are only stored in the MemStore. This is also stored in HDFS and is used for recovery in the case of a region server failure.
- **BlockCache**: this is the *read cache*. It stores frequently read data in memory and least recently used data is evicted when the cache is full.

As a result, write operations go through WAL and MemStore first and eventually end up being stored in HFiles[8], as shown in Figure 7.8. Read operations have to read from both the MemStore, the BlockCache and the existing HFiles and merge the results. This can be quite inefficient, so there are several optimisations used. As mentioned previously, columns are grouped by their column family and stored separately. As a result, only the HFiles that contain the required column family need to be queried. All the entries in an HFile are stored in lexicographical order and they contain an index in the end of the file which can be kept in memory, so reads can find the required data without reading the whole file. Each HFile also contains the time range of the entries contained in it to avoid unnecessary reads of files that cannot contain the requested data. Bloom filters[9] are also used to reduce the number of HFiles that need to be read; these are special data structures that make it easy to identify whether some data is not contained in a file using a very small amount of memory. There is also a background process, called compaction, which merges multiple HFiles into a single HFile removing older versions of data that are not needed anymore, thus reducing the number of HFiles that need to be inspected during read operations.

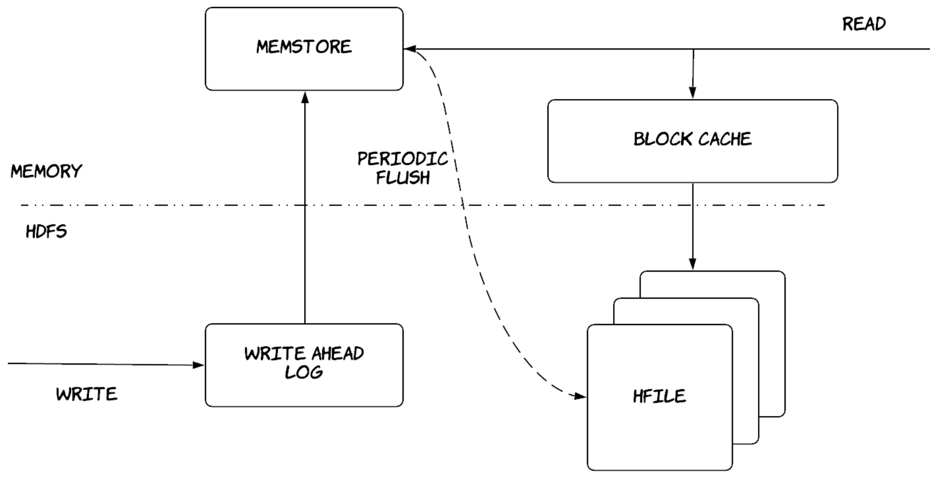


Figure 7.8: read and write data flow in HBase

Some of the guarantees provided by HBase are the following:

Atomicity:

[8]This pattern originates from a data structure, called a log-structured merge (LSM) tree[62]

[9]See: https://en.wikipedia.org/wiki/Bloom_filter

- Operations that mutate multiple rows are atomic.
  - An operation that returns a success code has completely succeeded.
  - An operation that returns a failure code has completely failed.
  - An operation that times out may have succeeded or may have failed. However, it cannot have partially succeeded or failed.
  - This is true even if the mutation crosses multiple column families within a row.[10]
- Operations that mutate multiple rows will not be atomic. For example, a mutative operation on rows 'a', 'b' and 'c' may return having mutated some but not all of the rows. In this case, the operation will return a list of codes, some of which may be successes, failures or timeouts.
- Hbase provides a conditional operation, called `checkAndPut`, which happens atomically like the typical `compareAndSet` (CAS) operation found in many hardware architectures.

Consistency & Isolation:

- Single-row reads/writes are **linearizable**.
  - When a client receives a successful response for any mutation, this mutation is immediately visible to both that client and any client with whom it later communicates through side channels.
- HBase provides a `scan` operation that provides efficient iteration over multiple rows. This operation does not provide a consistent view of the table and does not exhibit snapshot isolation. Instead:
  - Any row returned by the `scan` is a consistent view, i.e. that version of the complete row existed at some point in time.
  - A `scan` operation must reflect all mutations committed prior to the construction of the scanner and *may* reflect some mutations committed subsequent to the construction of the scanner.

Durability:

- All visible data is also durable. This means that a read will never return data that has not been made durable on disk.
- Any mutative operation that returns a successful response has been made durable.
- Any operation that has been made durable is stored in at least `n` different servers (Namenodes), where `n` is the configurable replication factor of HDFS.

[10]This is achieved by fine-grained, per-row locking. Note that HFiles are essentially immutable, so only the MemStore needs to participate in this which makes it very efficient.

As mentioned earlier, HBase and Bigtable have a very similar architecture with slightly different naming for the various components and different dependencies. The table below contains a mapping between HBase concepts and the associated concepts in Bigtable.

HBase	Bigtable
region	tablet
region server	tablet server
Zookeeper	Chubby
HDFS	GFS
HFile	SSTable
MemStore	Memtable

Cassandra

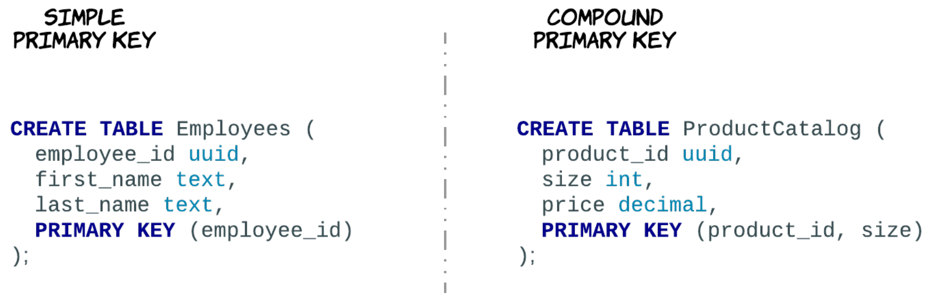
Cassandra is a distributed datastore that combined ideas from the Dynamo[8][11] and the Bigtable[7] paper. It was originally developed by Facebook[9], but it was then open sourced and became an Apache project. During this period, it has evolved significantly from its original implementation[12]. The main design goals of Cassandra are extremely high availability, performance (high throughput/low latency with emphasis on write-heavy workloads) with unbounded, incremental scalability. As explained later, in order to achieve these goals it trades off some other properties, such as strong consistency.

The data model is relatively simple: it consists of *keyspaces* at the highest level, which can contain multiple, different *tables*. Each table stores data in sets of *rows* and is characterised by a *schema*. This schema defines the structure of each row, which consists of the various columns and their types. The schema also determines the *primary key*, which is a column or a set of columns that have unique values for each row. The **primary key** can have two components: the first component is the **partition key** and it's mandatory, while the second component contains the **clustering columns**

[11]There is also a separate distributed system, which is called DynamoDB. This is commercially available, but details around its internal architecture have not been shared publicly yet. However, this system has a lot of similarities with Cassandra, such as the data model and tunable consistency.

[12]The information presented in this section refers to the state of this project at the time of writing.

and is optional. If both of these components are present, then the primary key is called a *compound primary key*. Furthermore, if the partition key is composed of multiple columns, it's called a *composite partition key*. Figure 7.9 contains an example of two tables, one having a simple primary key and one having a compound primary key.



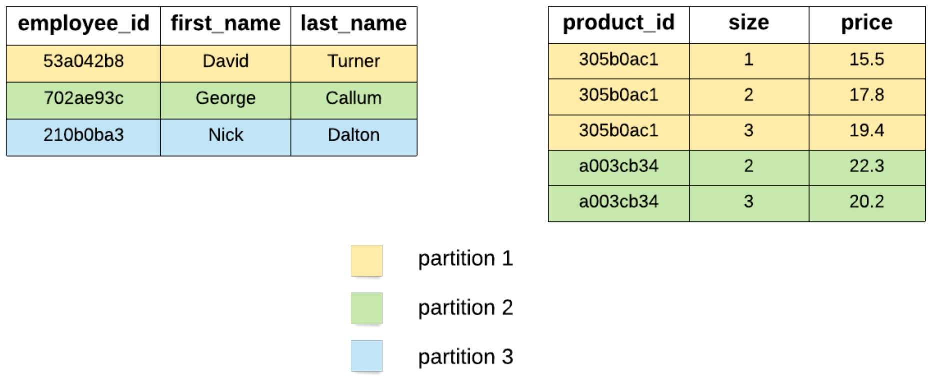
```
CREATE TABLE Employees (
  employee_id uuid,
  first_name text,
  last_name text,
  PRIMARY KEY (employee_id)
);
```

```
CREATE TABLE ProductCatalog (
  product_id uuid,
  size int,
  price decimal,
  PRIMARY KEY (product_id, size)
);
```

Figure 7.9: Cassandra data model

The primary key of a table is one of the most important parts of the schema, because it determines how data is distributed across the system and also how it is stored in every node. The first component of the primary key, the partition key determines the distribution of data. The rows of a table are conceptually split into different **partitions**, where each partition contains only rows with the same value for the defined partition key. All the rows corresponding to a single partition are guaranteed to be stored collocated in the same nodes, while rows belonging to different partitions can be distributed across different nodes. The second component of the primary key, the clustering columns, determine how rows of the same partition will be stored on disk. Specifically, rows of the same partition will be stored in ascending order of the clustering columns defined, unless specified otherwise. Figure 7.10 elaborates on the previous example, showing how data from the two tables would be split into partitions and stored in practice.

Cassandra distributes the partitions of a table across the available nodes using **consistent hashing**, while also making use of virtual nodes to provide balanced, fine-grained partitioning. As a result, all the virtual nodes of a Cassandra cluster form a ring. Each virtual node corresponds to a specific value in the ring, called the **token**, which determines which partitions will belong to this virtual node. Specifically, each virtual node contains all the partitions whose partition key (when hashed) falls in the range between



employee_id	first_name	last_name
53a042b8	David	Turner
702ae93c	George	Callum
210b0ba3	Nick	Dalton

product_id	size	price
305b0ac1	1	15.5
305b0ac1	2	17.8
305b0ac1	3	19.4
a003cb34	2	22.3
a003cb34	3	20.2

🟨	partition 1
🟩	partition 2
🟦	partition 3

Figure 7.10: Cassandra partitioning

its token and the token of the previous virtual node in the ring[13]. Every Cassandra node can be assigned multiple virtual nodes. Each partition is also replicated across `N` nodes, where `N` is a number that is configurable per keyspace and it's called the **replication factor**. There are multiple, available replication strategies that determine how the additional `N-1` nodes are selected. The simplest strategy just selects the next nodes clockwise in the ring. More complicated strategies also take into account the network topology of the nodes for the selection. The storage engine for each node is inspired by Bigtable and is based on a commit log containing all the mutations and a memtable that is periodically flushed to SSTables, which are also periodically merged via compactions.

The nodes of the cluster communicate with each other periodically via a gossip protocol, exchanging state and topology information about themselves and other nodes they know about. New information is gradually spread throughout the cluster via this process. In this way, nodes are able to keep track of which nodes are responsible for which token ranges, so that they can route requests accordingly. They can also determine which nodes are healthy and which are not, so that they can omit sending requests to nodes that are unreachable. Administrator tools are available that can be used by an operator to instruct a node of the cluster to remove another node that has crashed permanently from the ring. Any partitions belonging to

[13]Cassandra also supports some form of range partitioning, via the `ByteOrderedPartitioner`. However, this is available mostly for backwards compatibility reasons and it's not recommended, since it can cause issues with hot spots and imbalanced data distribution.

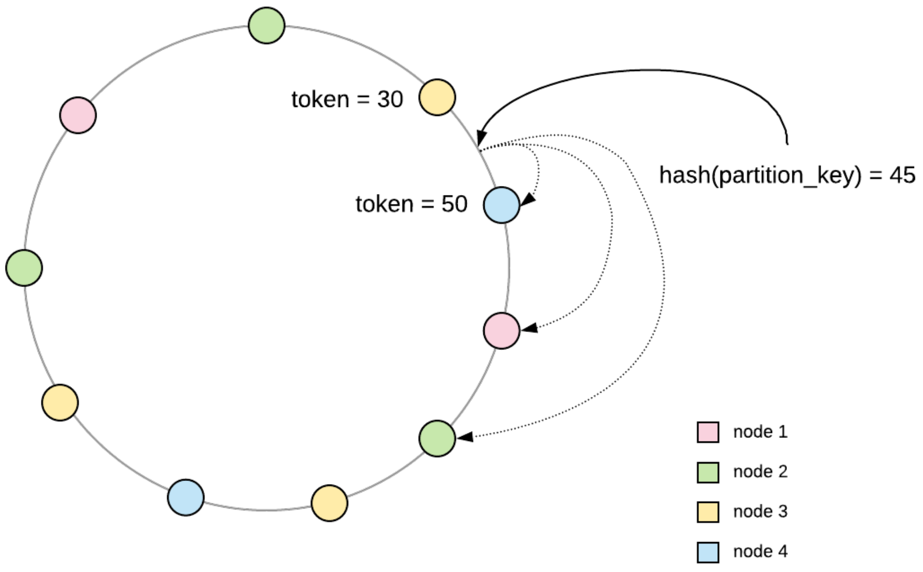


Figure 7.11: Cassandra partitioning

that node will be replicated to a different node from the remaining replicas. There is need for a bootstrap process that will allow the first nodes to join the cluster. For this reason, a set of nodes are designated as **seed nodes** and they can be specified to all the nodes of the cluster via a configuration file or a third-party system during startup.

Cassandra has no notion of a leader or primary node. All replica nodes are considered equivalent. Every incoming request can be routed to any node in the cluster. This node is called the **coordinator node** and is responsible for managing the execution of the request on behalf of the client. This node identifies the nodes that contain the data for the requested partition and dispatches the requests. After successfully collecting the responses, it replies to the client. Given there is no leader and all replica nodes are equivalent, they can be handling writes concurrently. As a result, there is a need for a conflict resolution scheme and Cassandra makes use of a last-write-wins (LWW) scheme. Every row that is written comes with a timestamp. When a read is performed, the coordinator collects all the responses from the replica nodes and returns the one with the latest timestamp.

The client can also specify policies that define how this coordinator node is selected. This policy might select coordinator nodes randomly in a round-robin fashion, select the closest node or select one of the replica nodes to reduce subsequent network hops. Similar to the concept of seed nodes, the client driver is provided some configuration that contains a list of *contact points*, which are nodes of the cluster. The client will initially try and connect to one of these nodes in order to acquire a view of the whole cluster and be able to route requests everywhere.

When communicating with Cassandra nodes, clients can specify **different consistency levels**, which allows them to optimise for consistency, availability or latency accordingly. The client can define the desired read consistency level and the desired write consistency level, where each consistency level provides different guarantees. Some of the available levels are the following:

- `ALL`: A write must be written to all replica nodes in the cluster for the associated partition. A read returns the record only after all replicas have responded, while the operation fails if a single replica does not respond. This option provides the highest consistency and the lowest availability.
- `QUORUM`: A write must be written on a quorum of replica nodes across all datacenters. A read returns the record after a quorum of replicas from all datacenters has replied. This option provides a balance between strong consistency and tolerance to a small number of failures.
- `ONE`: A write must be written to at least one replica node. A read returns the record after the response of a single replica node. This option provides the highest availability, but incurs a risk of reading stale data since the replica that replied might not have received the latest write.

The two consistency levels are not independent, so one should consider the interactions between them when deciding the appropriate level. If we assume a keyspace with replication factor `N` and clients that read with read consistency `R` and write with write consistency `W`, then a read operation is guaranteed to *reflect* the latest successful write as long as `R + W > N`. For instance, this could be achieved by performing both reads and writes at `QUORUM` level. Alternatively, it could be achieved by performing reads at `ONE` level and writes at `ALL` level or vice versa. In all of these cases, at least one node from the read set will exist in the write set, thus having seen the latest write. However, each one of them provides different levels of availability, durability, latency and consistency for read and write operations.

As illustrated so far, Cassandra favours high availability and performance over data consistency. As a result, it employs several mechanisms that ensure the cluster can keep processing operations even during node failures and partitions and the replicas can converge again as quickly as possible after recovery. Some of these mechanisms are the following:

- hinted handoff
- read repair
- anti-entropy repair

Hinted handoff happens during write operations: if the coordinator cannot contact the necessary number of replicas, then the coordinator can store locally the result of the operation and forward it to the failed node after it has recovered. *Read repair* happens during read operations: if the coordinator receives conflicting data from the contacted replicas, it resolves the conflict by selecting the latest record and forwards it synchronously to the stale replicas before responding to the read request. *Anti-entropy repair* happens in the background: replica nodes exchange the data for a specific range and if they find differences, they keep the latest data for each record, complying with the LWW strategy. However, this involves big datasets, so it's important to minimise consumption of network bandwidth. For this reason, the nodes encode the data for a range in a Merkle tree and exchange parts of the tree gradually, so that they can discover the conflicting data that need to be exchanged.

It is important to note that by default operations on a single row are **not linearizable** even when using majority quorums. To understand why, it is useful to understand how **partial failures** and **network delays** are handled by Cassandra. Let's examine two different scenarios.

- First, let's assume that read repair is not used. The system consists of 3 different replicas with a single row that contains a single column `owner` with value "none". Client A initially performs a write operation to set `owner = A`. While this operation is in progress, two different clients B and C perform a read operation for `owner` in sequence. The majority quorum of client B contains one replica that has already received the write operation, while client C contacts a quorum with nodes that haven't received it yet. As a result, client B reads `owner = A`, while client C reads `owner = none` even though the operation from the latter started after the operation from the former had completed, which violates linearizability. Figure 7.12 contains a diagram illustrating this phenomenon.

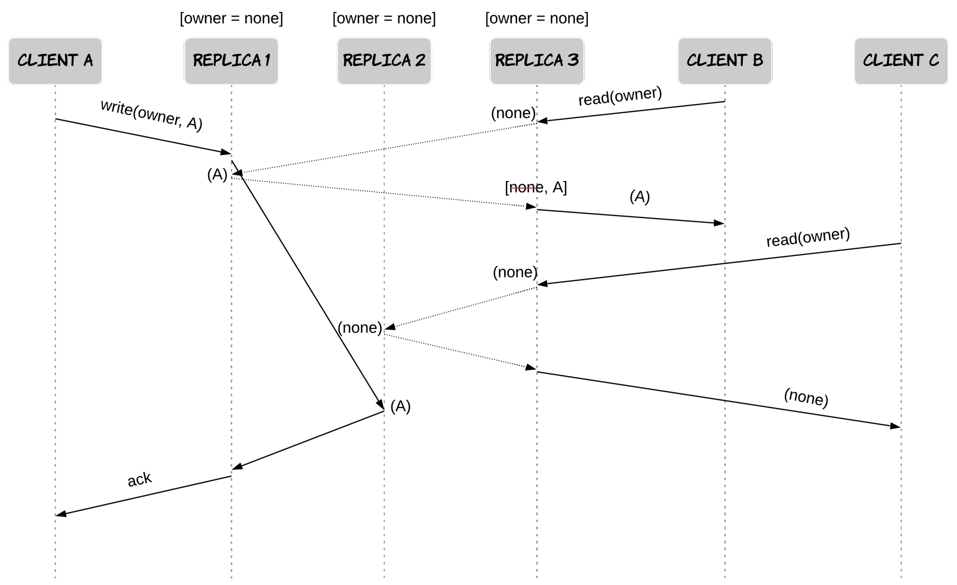


Figure 7.12: Linearizability violations when read repair is not used

- The violation of linearizability in the previous example would be eliminated if read repair was used, since the read from client B would propagate the value to the replica 2 and client C would also read `owner = A`. So, let's assume that read repair is used and examine a different scenario. Client A performs again a write operation to set `owner = A`. The write succeeds in one replica and fails in the other replica. As a result, the write is considered unsuccessful and the coordinator returns a failure response back to the client. Afterwards, client B performs a read operation that uses a quorum that contains the replica where the previous write succeeded. Cassandra performs a read repair using the LWW strategy, thus propagating the value to replica 2. As a consequence, a write operation that failed has affected the state of the database, thus violating linearizability. This example is shown in Figure 7.13.

Cassandra provides another consistency level that provides **linearizability** guarantees. This level is called `SERIAL` and the read/write operations executed in this level are also referred to as lightweight transactions. This level is implemented using a 4-phase protocol based on Paxos, as shown in Figure 7.14. The first and third phases of the protocol are the exact phases

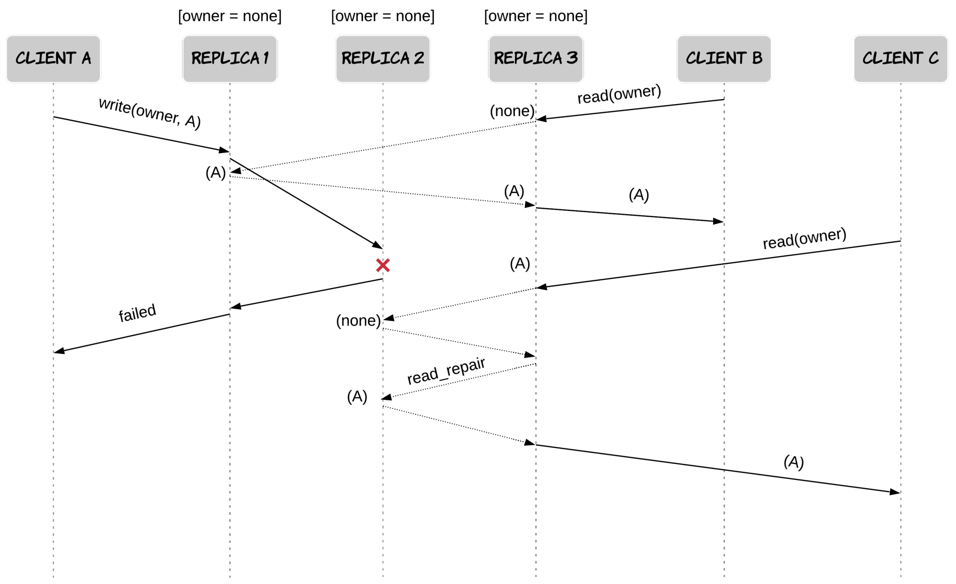


Figure 7.13: Linearizability violations when read repair is used

of Paxos and satisfy the same needs: the first phase is called `prepare` and corresponds to the nodes trying to gather votes before proposing a value which is done in the third phase, called `propose`. When run under `SERIAL` level, the write operations are conditional using an `IF` clause, also known as compare-and-set (CAS). The second phase of the protocol is called `read` and is used to retrieve the data in order to check whether the condition is satisfied before proceeding with the proposal. The last phase is called `commit` and it's used to move the accepted value into Cassandra storage and allow a new consensus round, thus unblocking concurrent LWTs again. Read and write operations executed under `SERIAL` are guaranteed to be linearizable. Read operations will commit any accepted proposal that has not been committed yet as part of the read operation. Write operations under `SERIAL` are required to contain a conditional part.

In Cassandra, performing a query that does not make use of the primary key is guaranteed to be inefficient, because it will need to perform a full table scan querying all the nodes of the cluster. There are two alternatives to this: *secondary indexes* and *materialized views*. A secondary index can be defined on some columns of a table. This means each node will index locally this table using the specified columns. A query based on these columns will still need

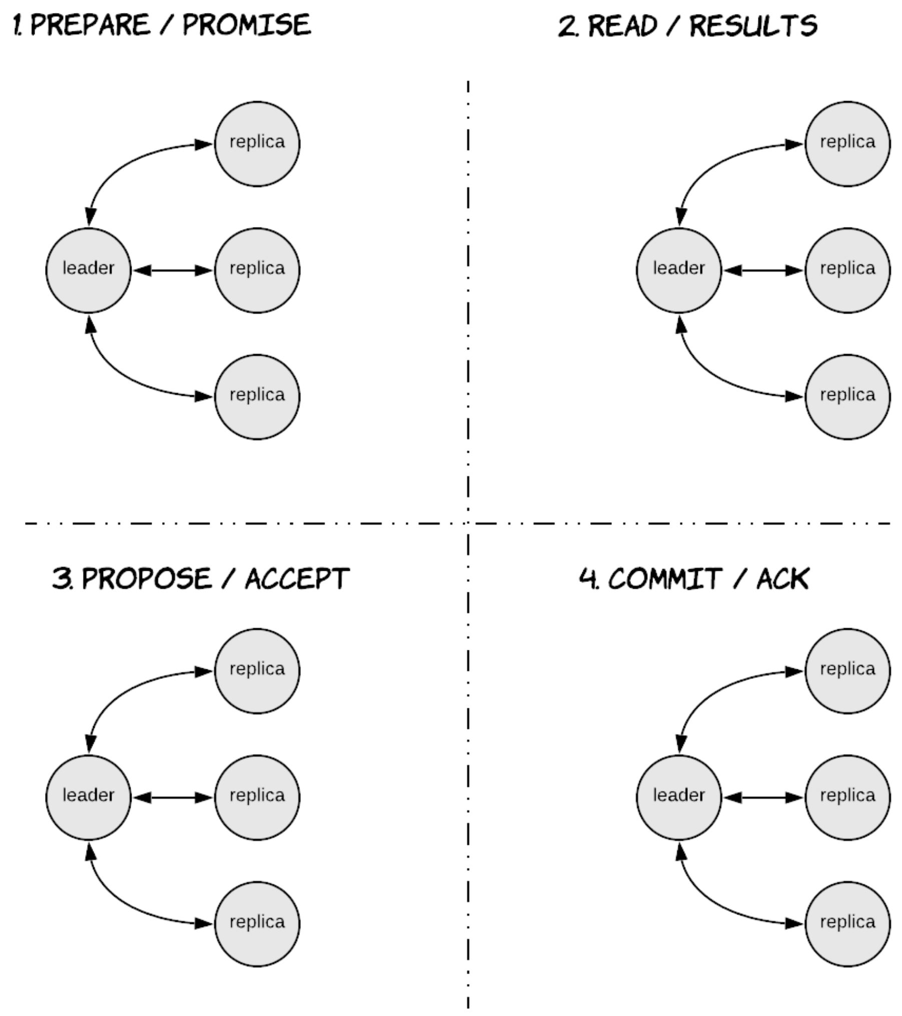


Figure 7.14: Phases of Cassandra LWT protocol

to ask all the nodes of the system, but at least each node will have a more efficient way to retrieve the necessary data without scanning all the data. A materialised view can be defined as a query on an existing table with a newly defined partition key. This materialised view is maintained as a separate table and any changes on the original table are eventually propagated to it. As a result, these two approaches are subject to the following trade-off:

- Secondary indexes are more suitable for high **cardinality** columns, while materialized views are suitable for low cardinality columns as they are stored as regular tables.
- Materialised views are expected to be more **efficient** during read operations when compared to secondary indexes, since only the nodes that contain the corresponding partition are queried.
- Secondary indexes are guaranteed to be **strongly consistent**, while materialised views are **eventually consistent**.

Cassandra does not provide join operations, since they would be inefficient due to the distribution of data. As a result, users are encouraged to denormalise the data by potentially including the same data in multiple tables, so that they can be queried efficiently reading only from a minimum number of nodes. This means that any update operations on this data will need to update multiple tables, but this is expected to be quite efficient. Cassandra provides 2 flavours of batch operations that can update multiple partitions and tables: **logged** and **unlogged** batches. Logged batches provide the additional guarantee of **atomicity**, which means either all of the statements of the batch operation will take effect or none of them. This can help ensure that all the tables that share this denormalised data will be consistent with each other. However, this is achieved by first *logging* the batch as a unit in a system table which is replicated and then performing the operations, which makes them less efficient than unlogged batches. Both logged and unlogged batches do not provide any isolation, so concurrent requests might observe the effects of some of the operations only temporarily.

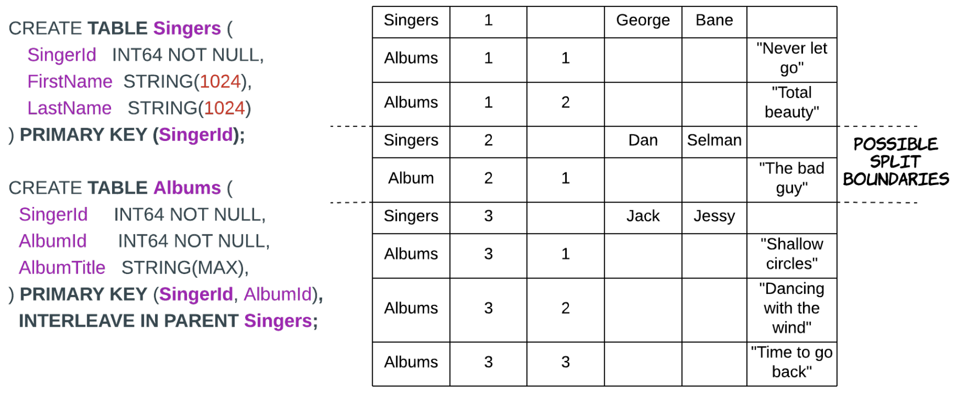
Spanner

Spanner is a distributed datastore that was initially developed internally by Google [5][63] and was subsequently released publicly as part of the Google platform[14].

[14]See: https://cloud.google.com/spanner

The data model of Spanner is very close to the data model of classical relational databases. A database in Spanner can contain one or more tables, which can contain multiple rows. Each row contains a value for each column that is defined and one or more columns are defined as the primary key of the table, which must be unique for each row. Each table contains a schema that defines the data types of each column.

Spanner partitions the data of a table using **horizontal range partitioning**. The rows of a table are partitioned in multiple segments, called *splits*. A split is a range of contiguous rows, where the rows are ordered by the corresponding primary key. Spanner can perform *dynamic load-based splitting*, so any split that receives an extreme amount of traffic can be partitioned further and stored in servers that have less traffic. The user can also define *parent-child* relationships between tables, so that related rows from the tables are collocated making join operations much more efficient. A table `C` can be declared as a child table of `A`, using the `INTERLEAVE` keyword and ensuring the primary key of the parent table is a prefix of the primary key of the child table. An example is shown in Figure 7.15, where a parent table `Singers` is interleaved with a child table, called `Albums` . Spanner guarantees that the row of a parent table and the associated rows of the child table will never be assigned to a different split.


```
CREATE TABLE Singers (
  SingerId   INT64 NOT NULL,
  FirstName  STRING(1024),
  LastName   STRING(1024)
) PRIMARY KEY (SingerId);

CREATE TABLE Albums (
 SingerId     INT64 NOT NULL,
 AlbumId      INT64 NOT NULL,
 AlbumTitle   STRING(MAX),
) PRIMARY KEY (SingerId, AlbumId),
 INTERLEAVE IN PARENT Singers;
```

Singers	1		George	Bane	
Albums	1	1			"Never let go"
Albums	1	2			"Total beauty"
Singers	2		Dan	Selman	
Album	2	1			"The bad guy"
Singers	3		Jack	Jessy	
Albums	3	1			"Shallow circles"
Albums	3	2			"Dancing with the wind"
Albums	3	3			"Time to go back"

Figure 7.15: Definition of interleaved tables and data layout

A Spanner deployment is called a *universe* and it consists of a set of *zones*, which are the units of administrative deployment, physical isolation and replication (e.g. datacenters). Each zone has a *zonemaster* and hundreds to several thousands *spanservers*. The former is responsible for assigning data to spanservers, while the latter process read/write requests from clients and store data. The per-zone *location proxies* are used by clients to locate the spanservers that serve a specific portion of data. The *universe master* displays status information about all the zones for troubleshooting and the *placement driver* handles automated movement of data across zones, e.g. for load balancing reasons.

Each spanserver can manage multiple splits and each split is replicated across multiple zones for availability, durability and performance[15]. All the replicas of a split form a **Paxos group**. One of these replicas is voted as the *leader* and is responsible for receiving incoming write requests and replicating them to the replicas of the group via a Paxos round. The rest of the replicas are *followers* and can serve some kinds of read requests. Spanner makes use of long-lived leaders with time-based leader leases, which are renewed by default every 10 seconds. Spanner makes use of pessimistic concurrency control to ensure proper isolation between concurrent transactions, specifically **two-phase locking**. The leader of each replica group maintains a *lock table* that maps ranges of keys to lock states for this purpose[16]. Spanner also provides support for **distributed transactions** that involve multiple splits that potentially belong to different replica groups. This is achieved via **two-phase commit** across the involved replica groups. As a result, the leader of each group also implements a *transaction manager* to take part in the two-phase commit. The leaders of each group that take part are referred to as *participant leaders* and the follower replicas of each one of those groups are referred to as *participant slaves*. More specifically, one of these groups is chosen as the coordinator for the two-phase commit protocol and the replicas of this group are referred to as *coordinator leader* and *slaves* respectively.

Spanner makes use of a novel API to record time, called *TrueTime* [64], which was the key enabler for most of the consistency guarantees provided by Spanner. This API directly exposes clock uncertainty and nodes can wait out that uncertainty when comparing timestamps retrieved from different clocks. If the uncertainty gets large because of some failure, this will manifest as increased latency due to nodes having to wait longer periods. TrueTime represents time as a *TTInterval*, which is an interval *[earliest, latest]* with bounded time uncertainty. The API provides a method `TT.now()` that re-

[15]In fact, each split is stored in a distributed filesystem, called *Colossus* that is the successor of GFS, which already provides byte-level replication. However, Spanner adds another level of replication to provide the additional benefits of data availability and geographic locality.

[16]In practice, these locks are also replicated in the replicas of the group to cover against failures of the leader.

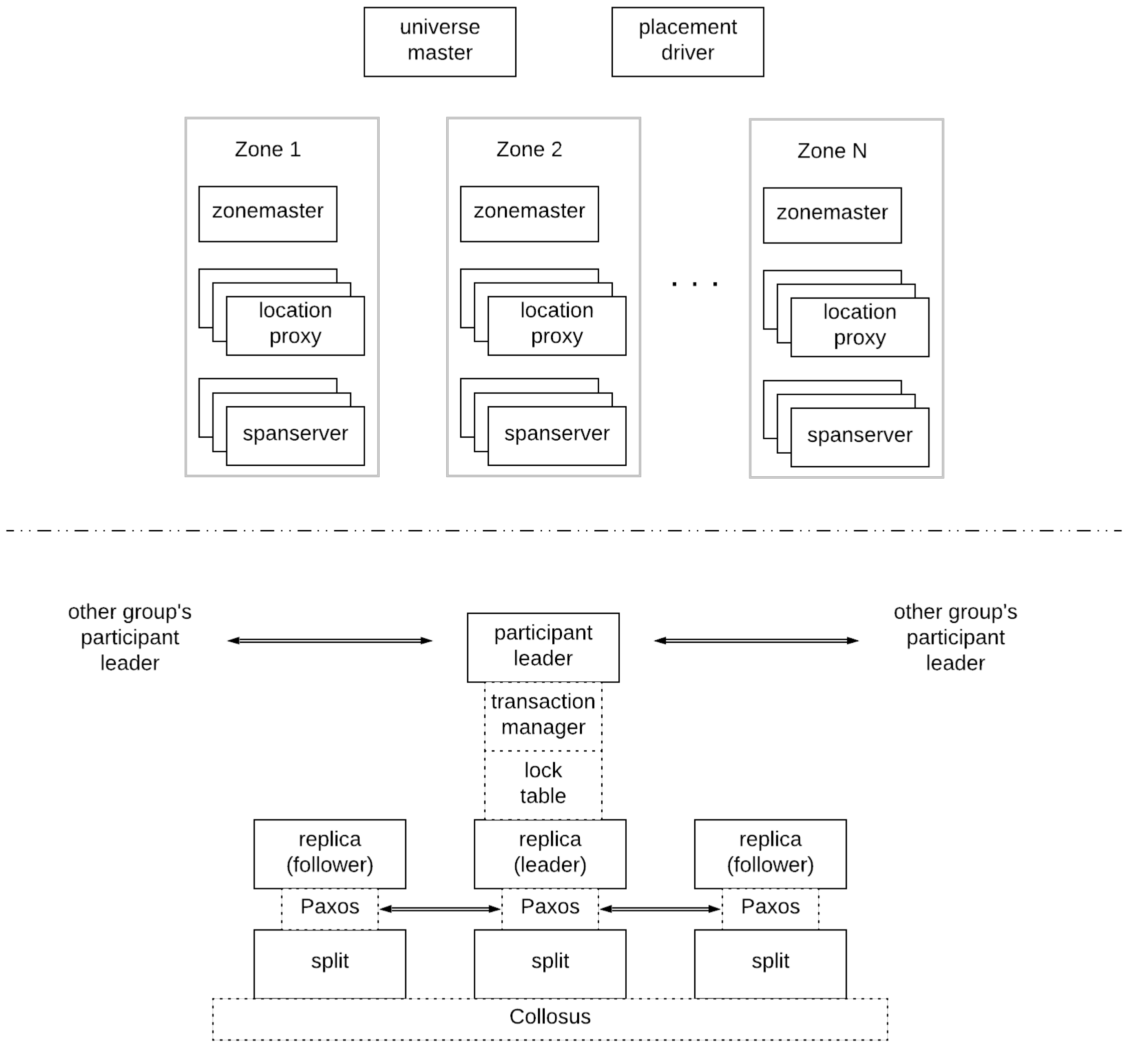


Figure 7.16: Spanner architecture

turns a *TTInterval* that is guaranteed to contain the absolute time during which the method was invoked[17]. It also provides two convenience methods `TT.after(t)`, `TT.before(t)` that specify whether `t` is definitely in the past or in the future. These are essentially just wrappers around `TT.now()`, since `TT.after(t) = t < TT.now().earliest` and `TT.before(t) = t > TT.now().latest`. As a result, Spanner can assign timestamps to transactions that have global meaning and can be compared by nodes having different clocks. TrueTime is implemented by a set of *time master* machines per datacenter and a *timeslave daemon* per machine. The masters can use one of two different forms of time reference, either GPS or atomic clocks, since they have different failure modes. The master servers compare their time references periodically and they also cross-check the rate at which their reference time advances against their local clock, evicting themselves from the cluster if there is a significant divergence. Daemons poll a variety of masters to synchronise their local clocks and advertise an uncertainty `e` which corresponds to half of the interval's width `(latest - earliest) / 2`. This uncertainty depends on master-daemon communication latency and the uncertainty of the masters' time. This uncertainty is a sawtooth function of time that is slowly increasing between synchronisations. In Google's production environment, the average value of this uncertainty was reported to be 4 milliseconds.

Spanner supports the following types of operations:

- *standalone (strong or stale) reads*
- *read-only transactions*
- *read-write transactions*

A read-write transaction can contain both read and/or write operations. It provides full **ACID** properties for the operations of the transaction. More specifically, read-write transactions are not simply serializable, but they are **strictly serializable**[18]. A read-write transaction executes a set of reads and write operations **atomically** at a single logical point in time. As explained before, Spanner achieves these properties with the use of **two-phase locking** for isolation and **two-phase commit** for atomicity across multiple splits. More specifically, the workflow is the following:

[17]As also explained in the chapter about time, this is assuming there's an idealized *absolute* time that uses the Earth as a single frame of reference and is generated using multiple atomic clocks. See: https://en.wikipedia.org/wiki/International_Atomic_Time

[18]In fact, Spanner documentation also refers to *strict serializability* with the name "*external consistency*", but both are essentially the same guarantees.

- After opening a transaction, a client directs all the read operations to the leader of the replica group that manages the split with the required rows. This leader acquires read locks for the rows and columns involved before serving the read request. Every read also returns the timestamp of any data read.
- Any write operations are buffered locally in the client until the point the transaction is committed. While the transaction is open, the client sends *keepalive* messages to prevent participant leaders from timing out a transaction.
- When a client has completed all reads and buffered all writes, it starts the two-phase commit protocol[19]. It chooses one of the participant leaders as the coordinator leader and sends a prepare request to all the participant leaders along with the identity of the coordinator leader. The participant leaders that are involved in write operations also receive the buffered writes at this stage.
- Every participant leader acquires the necessary write locks, chooses a *prepare* timestamp that is larger than any timestamps of previous transactions and logs a prepare record in its replica group through Paxos. The leader also replicates the lock acquisition to the replicas to ensure they will be held even in the case of a leader failure. It then responds to the coordinator leader with the prepare timestamp.
- The coordinator leader waits for the prepare response from all participant leaders. Afterwards, it acquires write locks and selects the commit timestamp of the transaction `s`. This must be greater or equal to all prepare timestamps from the participant leaders, greater than `TT.now().latest` at the time the coordinator received the commit request from the client and greater than any timestamps it has assigned to previous transactions. The coordinator leader then logs a commit record at its replica group through Paxos and then sends the commit timestamp to the client and all the participant leaders. In fact, the coordinator waits until `TT.after(s)` before doing that to ensure that clients cannot see any data committed by `s` until after `TT.after(s)` is true.
- Each participant logs the transaction's outcome through Paxos at its replica group, applies the transaction's writes at the commit timestamp and then release any locks held from this transaction.

[19]The two-phase commit is required only if the transaction accesses data from multiple replica groups. Otherwise, the leader of the single replica group can commit the transaction only through Paxos.

Figure 7.17 contains a visualisation of this sequence. It is worth noting that the availability problems from two-phase commit are partially mitigated in this scheme by the fact that both the participants and the coordinator are essentially a Paxos group. So, if one of the leader nodes crashes, then another replica from that replica group will eventually detect that, take over and help the protocol make progress. Furthermore, the two-phase locking protocol can result in deadlocks. Spanner resolves these situations via a wound-wait scheme [65], where a transaction TX_1 is allowed to abort a transaction TX_2 that holds a desired lock only if TX_1 is older than TX_2.

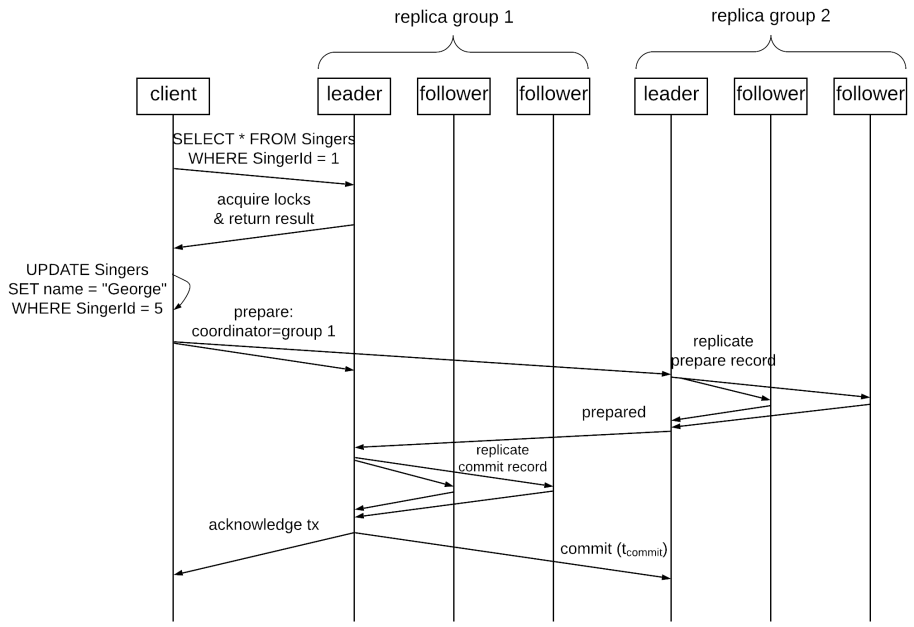


Figure 7.17: Read-write transactions in Spanner

Spanner needs a way to know if a replica is up-to-date to satisfy a read operation. For this reason, each replica tracks a value called *safe* time t_{safe}, which is the maximum timestamp at which the replica is up-to-date. Thus, a replica can satisfy a read at a timestamp `t` if $t \leq t_{safe}$. This value is calculated as $t_{safe} = \min({t_{safe}}^{Paxos}, {t_{safe}}^{TM})$. ${t_{safe}}^{Paxos}$ is the timestamp of the highest-applied Paxos write at a replica group and represents the highest watermark below which writes will no longer occur with respect to Paxos. ${t_{safe}}^{TM}$ is calculated as $\min_i({s_{i,g}}^{prepare})$ over all transactions T_i prepared (but not committed yet) at replica group g. If there is no such transactions, then

${t_{safe}}^{TM} = + \infty$.

Read-only transactions allow a client to perform multiple reads at the same timestamp and these operations are also guaranteed to be **strictly serializable**. An interesting property of read-only transactions is they do not need to hold any locks and they don't block other transactions. The reason for this is that these transactions perform reads at a specific timestamp, which is selected in such a way as to guarantee that any concurrent/future write operations will update data at a later timestamp. The timestamp is selected at the beginning of the transaction as `TT.now().latest` and it's used for all the read operations that are executed as part of this transaction. In general, the read operations at timestamp t_{read} can be served by any replica g that is up to date, which means $t_{read} \leq t_{safe,g}$. More specifically:

- In some cases, a replica can be certain via its internal state and TrueTime that it is up to date enough to serve the read and does so.
- In some other cases, a replica might not be sure if it has seen the latest data. It can then ask the leader of its group for the timestamp of the last transaction it needs to apply in order to serve the read.
- In the case the replica is the leader itself, it can proceed directly since it is always up to date.

Spanner also supports standalone reads outside the context of transactions. These do not differ a lot from the read operations performed as part of read-only transactions. For instance, their execution follows the same logic using a specific timestamp. These reads can be *strong* or *stale*. A strong read is a read at a current timestamp and is guaranteed to see all the data that has been committed up until the start of the read. A stale read is a read at a timestamp in the past, which can be provided by the application or calculated by Spanner based on a specified upper bound on staleness. A stale read is expected to have lower latency at the cost of stale data, since it's less likely the replica will need to wait before serving the request.

There is also another type of operations, called *partitioned DML*. This allows a client to specify an update/delete operation in a declarative form, which is then executed in parallel at each replica group. This parallelism and the associated data locality makes these operations very efficient. However, this comes with some tradeoffs. These operations need to be **fully partitionable**, which means they must be expressible as the union of a set of statements, where each statement accesses a single row of the table and each statement accesses no other tables. This ensures each replica group will be able to execute the operation locally without any coordination with other replica groups. Furthermore, these operations need to be **idempotent**, because Spanner might execute a statement multiple times against some groups due to network-level retries. Spanner does not provide **atomicity guarantees** for each statement across the entire table, but it provides atomicity guarantees per each group. This means that a statement might only run against some rows of the table, e.g. if the user cancels the operation midway or the execution fails in some splits due to constraint violations.

FaunaDB

FaunaDB[20] is a distributed datastore that drew inspiration from the Calvin protocol[66] for its core architecture. Calvin is based on the following central idea: by replicating inputs instead of effects to the various nodes of the system, it's possible to have a system that is more deterministic where all the non-failing nodes go through the same states. This determinism can obviate the need for agreement protocols, such as **two-phase commit**, when performing distributed transactions, since the nodes involved in the transaction can rely on each other proceeding in exactly the same way.

Abstractly, the architecture is composed of three layers:

- the **sequencing** layer: this is responsible for receiving inputs/commands and placing them in a global order, which is achieved via a consensus protocol. This is the sequence the operations will be executed by all the nodes.
- the **scheduling** layer: this is responsible for orchestrating the execution of transactions using a deterministic locking scheme to guarantee equivalence to the serial order specified by the sequencing layer, while also allowing transactions to be executed concurrently.
- the **storage** layer: this is responsible for the physical data layout.

In practice, every node in FaunaDB performs three roles simultaneously:

- **query coordinator**: this is responsible for receiving and processing a request. As explained later, the request might be processed locally or routed to other nodes, depending on its type.
- **data replica**: this is responsible for storing data and serving them during read operations

[20]See: https://fauna.com

- **log replica**: this is responsible for reaching consensus on the order of inputs and adding them to the globally ordered log.

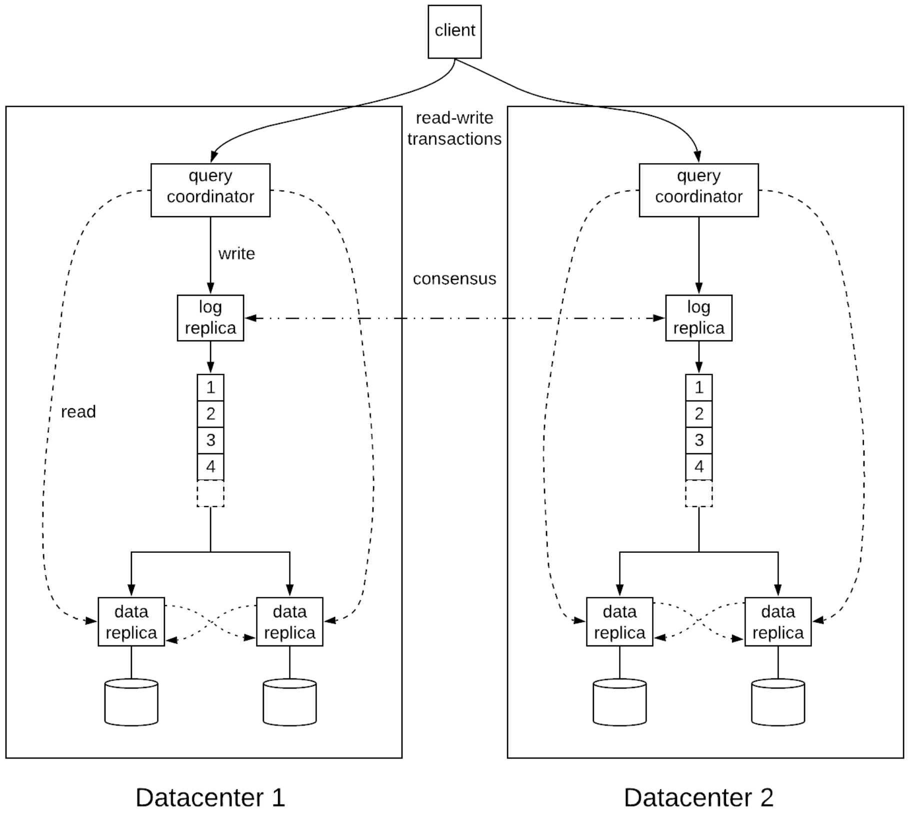


Figure 7.18: Conceptual view of FaunaDB's architecture

A cluster is made up of three or more logical datacenters and data is **partitioned** inside a datacenter and **replicated** across datacenters for increased **performance** and **availability**. Multiple versions of each data item are preserved, similar to Spanner. FaunaDB uses a slightly customised version of Raft for consensus, which aggregates requests and replicates them in batches[21] to improve throughput. When a request arrives to a query coordinator, it speculatively executes the transaction at the latest known log

[21]These batches are called *epochs* and a typical window of batching is 10 milliseconds, so that the impact on latency is not significant. The ordering of requests is achieved by combining the epoch number and the index of the request in the batch.

timestamp to discover the data accessed by the transaction, also referred to as read and write *intents*. The processing thereafter differs depending on the type of request:

- If the request is a read-write transaction, it is forwarded to a log replica that makes sure it's recorded as part of the next batch, as agreed via consensus with the other replicas. The request is then forwarded to each data replica that contains associated data. An interesting difference with other systems is that in FaunaDB data transfer at this stage is **push-based**, not **pull-based**. As an example, if during a transaction replica A needs to perform a write based on data owned by replica B, replica B is supposed to send the data to replica A, instead of replica A requesting them[22]. As a result, each data replica blocks until it has received all the data needed from other replicas. Then, it resolves the transaction, applies any local writes and acknowledge the success to the query coordinator. It's important to note that data might have changed since the speculative execution of the query coordinator. If that's the case, the transaction will be aborted and can potentially be retried, but this will be a unanimous decision since all the nodes will execute the operations in the same order. As a consequence, there is no need for an agreement protocol, such as two-phase commit.
- If the request is a read-only transaction, it is sent to the replica(s) that contain the associated data or served locally, if the query coordinator happens to contain all the data. The transaction is timestamped with the latest known log timestamp and all read operations are performed at this timestamp. The client library also maintains the timestamp of the highest log position seen so far, which is used to guarantee a monitonically advancing view of the transaction order. This guarantees **causal consistency** in cases where the client switches from node A to node B, where node B is lagging behind node A in transaction execution from the log.

In terms of guarantees, read-write transactions are **strictly serializable** and read-only transactions are only **serializable**. However, read-only transactions can opt-in to be strictly serializable by using the so-called `linearized` endpoint. In that case, the read is combined with a no-op write and it's executed as a regular read-write transaction going through consensus, thus

[22]A significant advantage of this is fewer messages that lead to reduced latency. A valid question is what happens if the node that is supposed to send the data fails. In this case, the data replica can fall back to requesting the data from other replicas of this partition.

taking a latency hit.

To achieve these guarantees, read-write transactions make use of a pessimistic concurrency control scheme based on read/write locks. This protocol is deterministic, which means it guarantees that all nodes will acquire and release locks in the exact, same order[23]. This order is defined by the order of the transactions in the log. Note that this does not prevent transactions from running concurrently, it just requires that locks for transaction t_i can be acquired only after locks have been acquired (and potentially released) for all previous transactions t_j $(j < i)$.

This means that all the data accessed by read/write transactions need to be known in advance[24], which means FaunaDB cannot support interactive transactions. Interactive transactions are ones that a client can keep open and execute operations dynamically while potentially performing other operations not related to the database. In contrast to that, transactions in FaunaDB are declared at once and sent for processing. Interactive transactions could still be simulated via a combination of snapshot reads and compare-and-swap operations.

Figure 7.18 contains a conceptual view of the architecture described so far. Each role is visualised separately in the figure to facilitate understanding of how the various functions interoperate. However, a single node can perform all these roles, as explained previously.

Distributed messaging system (Kafka)

Apache Kafka is an open-source messaging system initially developed by Linkedin[68][69] and then donated to the Apache Software Foundation[25]. The primary goal of Kafka was:

- **performance**: the ability to exchange messages between systems with high throughput and low latency.

[23]An interesting benefit of this is that deadlocks are prevented. There is literature that examines in more detail the benefits of determinism in database systems[67].

[24]As described previously, this is not strictly required, since the query coordinator performs an initial *reconaissance* query and includes the results in the submitted transactions. So, all the replicas can perform again the reads during the execution of the transaction and identify whether read/write sets have changed, where the transaction can be aborted and retried. This technique is called Optimistic Lock Location Prediction (OLLP).

[25]See: https://kafka.apache.org/

- **scalability**: the ability to incrementally scale to bigger volumes of data by adding more nodes to the system.
- **durability & availability**: the ability to provide durability and availability of data even in the presence of node failures.

The central concept of Kafka is the **topic**. A topic is an ordered collection of **messages**. For each topic, there can be **multiple producers** that write messages to it. There can also be **multiple consumers** that read messages from it[26]. To achieve performance and scalability, each topic is maintained as a **partitioned log**, which is stored across multiple nodes called **brokers**. Each partition is an ordered, immutable sequence of messages, where each message is assigned a sequential id number called the **offset**, which uniquely identifies each message within the partition. Messages by producers are always appended to the end of the log. Consumers can consume records in any order they like providing an offset, but normally a consumer will advance its offset linearly as it reads records. This provides some useful flexibility, which allows consumers to do things like replaying data starting from an older offset or skipping messages and start consuming from the latest offset. The messages are stored durably by Kafka and retained for a configurable amount of period, called **retention period**, regardless of whether they have been consumed by some client.

As explained before, every log is partitioned across multiple servers in a Kafka cluster. Messages written by producers are distributed across these partitions. This can be done in a round-robin fashion simply to balance load or the partition can be selected by the producer according to some semantic partitioning function (e.g. based on some attribute in the message and a partitioning function), so that related messages are stored in the same partition. Each consumer of a topic can have multiple consumer instances for increased performance, which are all identified by a **consumer group** name. Consumption is implemented in such a way that partitions in a log are divided over the consumer instances, so that each instance is the exclusive consumer of a *"fair share"* of partitions. As a result, each message published to a topic is delivered to one consumer instance within each subscribing consumer group. Each partition is also **replicated** across a configurable number of nodes for **fault tolerance**. Each partition has one node which acts as the **leader** and zero or more servers that act as **followers**. The leader handles all read and write requests for the partition, while the followers

[26]This means Kafka can support both the *point-to-point* and the *publish-subscribe* model depending on the number of consumers used.

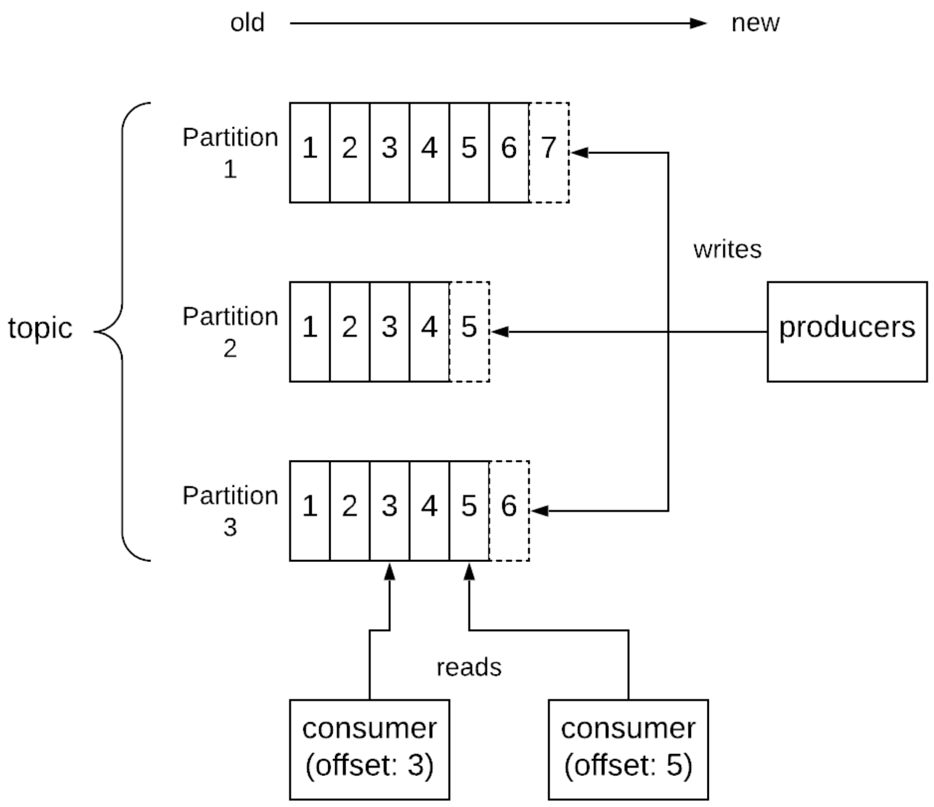


Figure 7.19: Structure of a Kafka topic

passively replicate the leader. If the leader fails, one of the followers will automatically detect that and become the new leader.

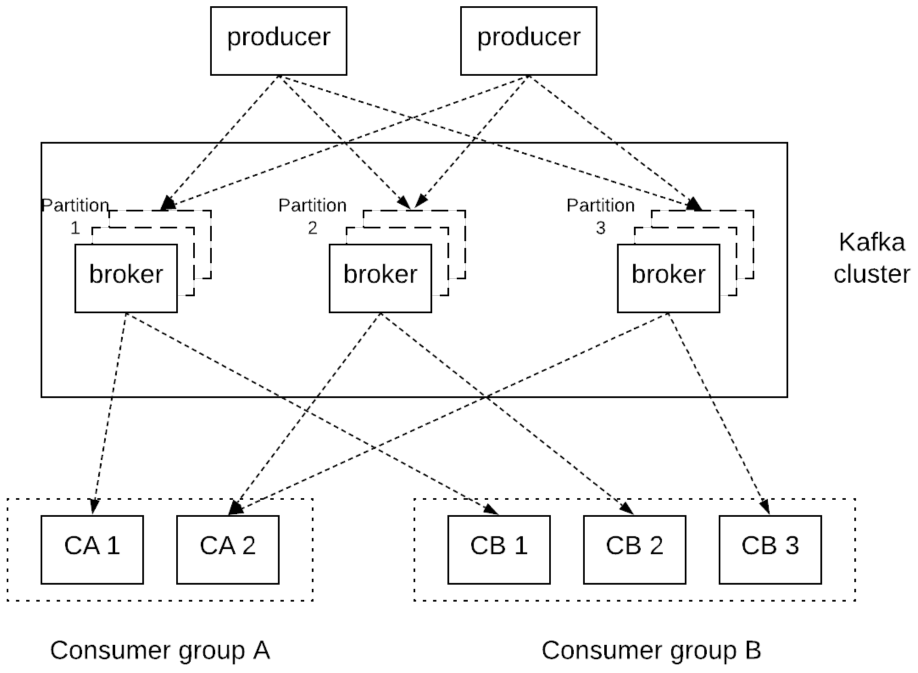


Figure 7.20: Kafka architecture

Kafka makes use of Zookeper for various functions, such as leader election between the replica brokers and group membership of brokers and consumers. Interestingly, log replication is separated from the key elements of the consensus protocol, such as leader election and membership changes. The latter are implemented via Zookeper, while the former is using a **single-master** replication approach, where the leader waits for followers to persist each message before acknowledging it to the client. For this purpose, Kafka has the concept of in-sync replicas (ISR), which are replicas that have replicated committed records and are thus considered to be in-sync with the leader. In case of a leader failure, only a replica that is in the ISR set is allowed to be elected as a leader. This guarantees **zero data loss**, since any replica in the ISR set is guaranteed to have stored locally all the records acknowledged by the previous leader. If a follower in the ISR set is very slow and lags behind, the leader can evict that replica from the ISR set in order to make progress. In this case, it's important to note that the ISR update is completed before proceeding, e.g. acknowledging records that have been persisted by the new, smaller ISR set. Otherwise, there would be a risk of data loss, if the leader failed after acknowledging these records but before updating the ISR set, so that the slow follower could be elected as the new leader even though it would be missing some acknowledged records. The leader maintains 2 offsets, the log end offset (LEO) and the high watermark (HW). The former indicates the last record stored locally, but not replicated or acknowledged yet. The latter indicates the last record that has been successfully replicated and can be acknowledged back to the client.

Kafka provides a lot of levers to adjust the way it operates depending on the application's needs. These levers should be tuned carefully depending on requirements around **availability**, **durability** and **performance**. For example, the user can control the replication factor of a topic, the minimum size of the ISR set (`min.insync.replicas`) and the number of replicas from the ISR set that need to acknowledge a record before it's committed (`acks`). Let's see some of the trade-offs one can make using these values:

- Setting `min.insync.replicas` to a majority quorum (e.g. `(replication factor / 2) + 1`) and `acks` to `all` would allow one to enforce stricter durability guarantees, while also achieving good availability. Let's assume `replication factor = 5`, so there are 5 replicas per partition and `min.insync.replicas = 3`. This would mean up to 2 node failures can be tolerated with zero data loss and the cluster still being available for writes and reads.
- Setting `min.insync.replicas` equal to `replication factor` and `acks` to `all` would provide even stronger durability guarantees at the expense of lower availability. In our previous example of `replication factor = 5`, this would mean that up to 4 node failures can now be tolerated with zero data loss. However, a single node failure makes the cluster unavailable for writes.
- Setting `acks` to `1` can provide better performance at the expense of weaker durability and consistency guarantees. For example, records will be considered committed and acknowledged as soon as the leader has stored them locally without having to wait for any of the followers to catch up. However, in case of a leader failure and election of a new leader, records that had been acknowledged by the previous leader but had not made it to the new leader yet will be lost.

Kafka can provide at-least-once, at-most-once and exactly-once messaging guarantees through various different configurations. Let's see each one of them separately:

- **at-most-once** semantics: this can be achieved on the producer side by disabling any retries. If the write fails (e.g. due to a `TimeoutException`), the producer will not retry the request, so the message might or might not be delivered depending on whether it had reached the broker. However, this guarantees that the message cannot be delivered more than once. In a similar vein, consumers commit message offsets before they process them. In that case, each message is processed once in the happy path. However, if the consumer fails after committing the offset but before processing the message, then the message will never be processed.
- **at-least-once** semantics: this can be achieved by enabling retries for producers. Since failed requests will now be retried, a message might be delivered more than once to the broker leading to duplicates, but it's guaranteed it will be delivered at least once[27]. The consumer can process the message first and then commit the offset. This would mean that the message could be processed multiple times, if the consumer fails after processing it but before committing the offset.
- **exactly-once** semantics: this can be achieved using the **idempotent** producer provided by Kafka. This producer is assigned a unique identifier (PID) and tags every message with a sequence number. In this way, the broker can keep track of the largest number per PID and reject duplicates. The consumers can store the committed offsets in Kafka or in an external datastore. If the offsets are stored in the same datastore where the side-effects of the message processing are stored, then the offsets can be committed atomically with the side-effects, thus providing exactly-once guarantees.

Kafka also provides a transactional client that allows producers to produce messages to multiple partitions of a topic atomically. It also makes it possible to commit consumer offsets from a source topic in Kafka and produce messages to a destination topic in Kafka atomically. This makes it possible to provide exactly-once guarantees for an end-to-end pipeline. This is achieved through the use of a **two-phase commit** protocol, where the brokers of the cluster play the role of the transaction coordinator in a highly available manner using the same underlying mechanisms for partitioning, leader election and fault-tolerant replication. The coordinator stores the

[27]Note that this is assuming infinite retries. In practice, a maximum threshold of retries is usually performed, in which case a message might not be delivered if this limit is exhausted.

status of a transaction in a separate log. The messages contained in a transaction are stored in their own partitions as usual. When a transaction is committed, the coordinator is responsible for writing a commit marker to the partitions containing messages of the transactions and the partitions storing the consumer offsets. Consumers can also specify the isolation level they want to read under, `read_committed` or `read_uncommitted`. In the former case, messages that are part of a transaction will be readable from a partition only after a commit marker has been produced for the associated transaction. This interaction is summarised in Figure 7.21.

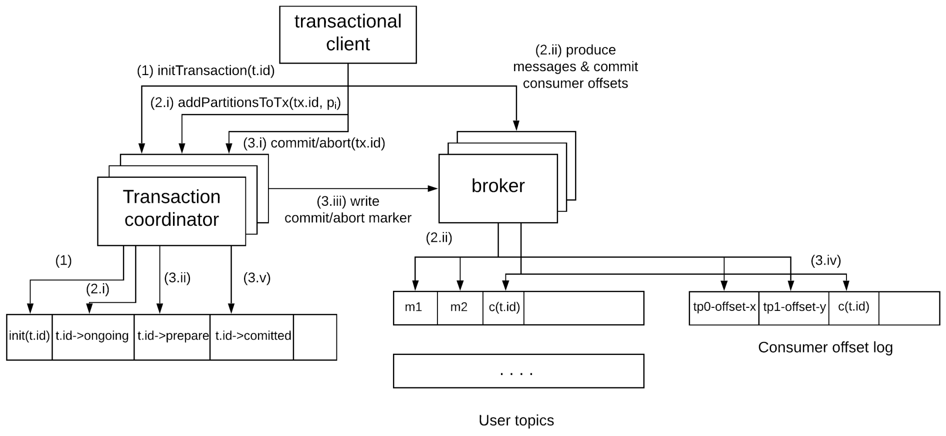


Figure 7.21: High-level overview of Kafka transactions

The physical storage layout of Kafka is pretty simple: every log partition is implemented as a set of segment files of approximately the same size (e.g. 1 GB). Every time a producer publishes a message to a partition, the broker simply appends the message to the last segment file. For better performance, segment files are flushed to disk only after a configurable number of messages have been published or a configurable amount of time has elapsed[28]. Each broker keeps in memory a sorted list of offsets, including the offset of the first message in every segment file. Kafka employs some more performance optimisations, such as using the `sendfile` API[29] for sending

[28]This behaviour is configurable through the values `log.flush.interval.messages` and `log.flush.interval.ms`. It is important to note that this behaviour has implications in the aforementioned durability guarantees, since some of the acknowledged records might be temporarily stored only in the memory of all in-sync replicas for some time until they are flushed to disk.

[29]See https://developer.ibm.com/articles/j-zerocopy

data to consumers thus minimizing copying of data and system calls.

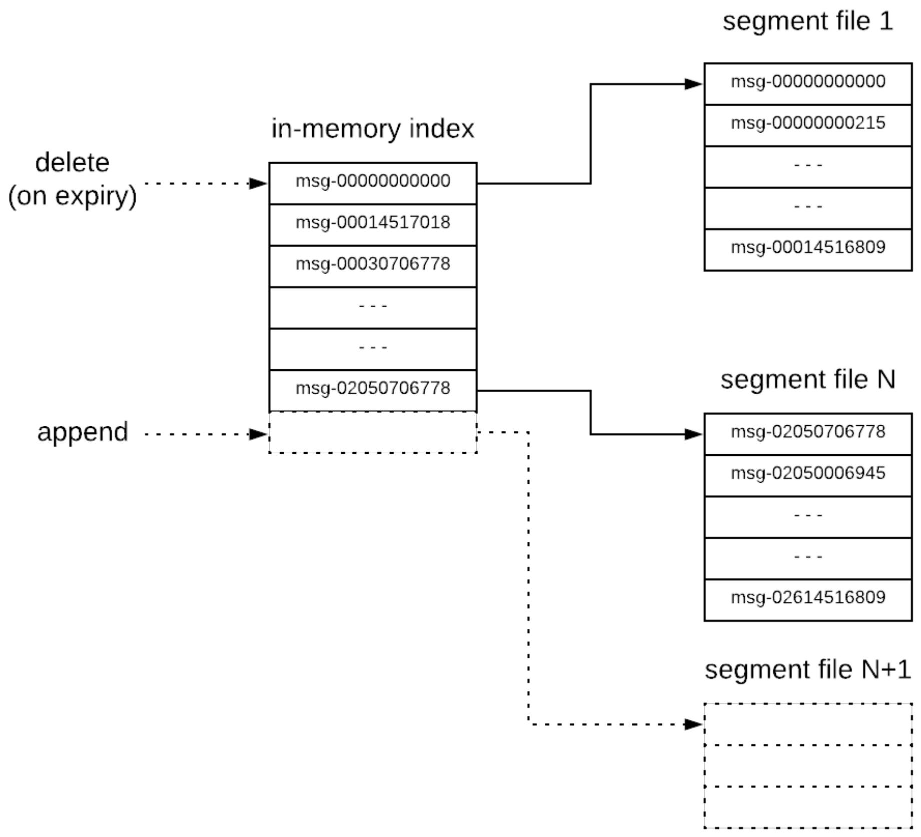


Figure 7.22: Storage layout of a Kafka topic

Some of the guarantees provided by Kafka are the following:

- Messages sent by a producer to a particular topic's partition will be appended in the order they are sent. That is, if a message M_1 is sent by the same producer as a message M_2, and M_1 is sent first, then M_1 will have a lower offset than M_2 and appear earlier in the log[30].
- As explained before, Kafka can provide at-least-once, at-most-once and exactly-once messaging semantics, depending on the configuration and the type of producers and consumers used.

[30]Note that ordering guarantees are provided only per partition. Users of Kafka can control partitioning, as described before, to leverage the ordering guarantees.

- The durability, availability and consistency guarantees provided by Kafka depend on the specific configuration of the cluster, as shown in the examples above. For example, a topic with `replication factor` of `N`, `min.insync.replicas` of `N/2 + 1` and `acks=all` guarantees zero data loss and availability of the cluster for up to N/2 failures.

Distributed cluster management (Kubernetes)

Kubernetes[31] is a system that was originally designed by Google, inspired by a similar system called Borg[70][71], and now maintained by the Cloud Native Computing Foundation. It can be used to manage a cluster of nodes and other resources (e.g. disks) handling all the aspects of running software in the cluster, such as deployment, scaling and discovery.

A Kubernetes cluster contains a set of nodes that can have 2 distinct roles, they can either be a **master** node or a **worker** node. A worker node is responsible for running the user applications. A master node is responsible for managing and coordinating the worker nodes. Essentially, worker nodes make a set of resources available to the cluster and master nodes decide how these resources are allocated to the applications that need to be executed as specified by the user[32]. For availability and durability, multiple master nodes can be run in parallel with one of them operating as the active **leader** and the rest acting as passive **followers**. Kubernetes uses etcd for various purposes, such as storing all the cluster data, performing leader election and transmitting change notifications between different parts of the cluster. Each node has several different components for the various functionalities that run independently, i.e. as different processes.

The various objects of the cluster (e.g. nodes, services, jobs etc.) are called *resources* and they are represented in etcd as key-value entries under the right namespace. One of the most central resources in Kubernetes is the *pod*, which represents the smallest deployable unit of computing. In practice, it is a group of one or more containers[33] with shared storage/network and a

[31]See: https://kubernetes.io/

[32]These applications can be divided in two main categories: *long-running* services that are supposed to be running constantly and typically respond to incoming requests and *jobs* that are supposed to run for a bounded amount of time typically doing some data processing.

[33]A container is a lightweight and portable executable image that contains software and all of its dependencies. Kubernetes supports multiple container runtimes with Docker

specification for how to run the containers. A *persistent volume* is a piece of storage in the cluster that has a lifecycle independent of any individual pod that uses it. A *job* creates one or more pods and ensures that a specified number of them successfully terminate. A *service* is an abstraction which defines a logical set of Pods and a policy by which to access them. Every resource is characterised by some *desired state* (e.g. number of replicas for a service) and the various components of Kubernetes cooperate to ensure the cluster's current state matches the desired state[34].

The main components of the master node are the API Server (`kube-apiserver`), the Scheduler (`kube-scheduler`) and the Controller Manager (`kube-controller-manager`). The API Server is essentially the front-end of the Kubernetes cluster allowing users to inspect the resources of the cluster and modify them or create new ones. The Scheduler is responsible for detecting newly created pods that have no node assigned and select a node for them to run. This selection is done based on multiple criteria, such as user-specified constraints, affinity specifications, data locality etc. The Controller Manager is responsible for running all the available *controllers* in the master node. A controller is essentially a control loop that watches the state of the cluster through the API server making changes in order to move the current state towards the desired state. Below are some examples of controllers:

- Node Controller: responsible for noticing and responding to node failures
- Replication Controller: responsible for maintaining the correct number pods according to the replication specified by the user.
- Endpoints Controller: responsible for creating endpoints for services

The main components of the worker nodes are the `kubelet` and the proxy (`kube-proxy`). The `kubelet` is an agent that runs on each node in the cluster, receives a set of pod specifications and makes sure the containers described in these specifications are running and are healthy. The proxy is a network proxy that maintains network rules that allow network communication to the pods from sessions inside and outside the cluster. The worker nodes also contain software of the container runtime that is used.

As a result, Kubernetes operates under eventual consistency, recovering from

being the most popular.

[34]The desired state is provided by the user when creating a resource (`Spec`), while the current state is supplied and updated by Kubernetes (`Status`).

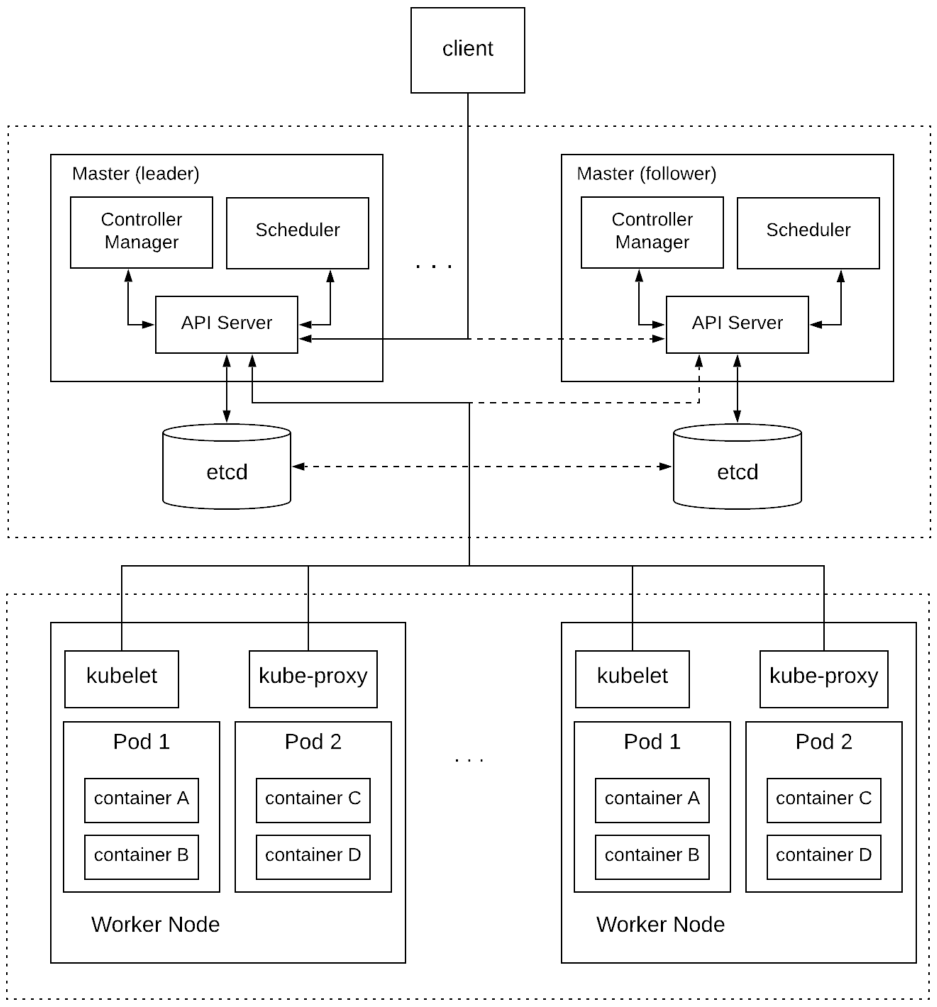


Figure 7.23: Kubernetes architecture

potential failures and converging back to the desired state. Since there are multiple components reading and updating the current state of the cluster, there is a need for some **concurrency control** to prevent anomalies arising from reduced isolation. Kubernetes achieves this with the use of conditional updates. Every resource object has a `resourceVersion` field representing the version of the resource as stored in etcd. This version can be used to perform a **compare-and-swap** (CAS) operation, so that anomalies like **lost updates** are prevented.

Distributed ledger (Corda)

Corda is a platform that allows multiple parties that do not fully trust each other to maintain a distributed ledger with shared facts amongst each other. By its nature, this means it is a distributed system similar to the systems analysed previously. However, a distinctive characteristic of this system is this lack of trust between the nodes that are part of the system, which also gives it a decentralisation aspect. This distrust is managed through various cryptographic primitives[35], as explained later. This section will give a rather brief overview of Corda's architecture, but you can refer to the available whitepapers for a more detailed analysis[72][73].

Each node in Corda is a JVM-runtime environment with a unique identity on the network. A Corda network is made up of many such nodes that want to transact with each other in order to maintain and evolve a set of shared facts. This network is *permissioned*, which means nodes need to acquire an X.509 certificate from the network operator in order to be part of the network. The component that issues these certificates is referred to as the *doorman*. In this context, the doorman operates as a certificate authority for the nodes that are part of the network. Each node maintains a public and a private key[36], where the private key is used to attest to facts by signing the associated data and the public key is used by other nodes to verify these signatures. This X.509 certificate creates an association between the public key of the node and a human-readable X.500 name (e.g. `O=MegaCorp,L=London,C=GB`). The

[35]This is a book about distributed systems, so this section will focus mostly on the distribution aspect of Corda. For the sake of completeness, the analysis might also mention how some cryptographic techniques are used, but this will be done under the assumption that the reader is familiar with basic concepts and can study them further outside the scope of this book.

[36]See: https://en.wikipedia.org/wiki/Public-key_cryptography

network also contains a *network map service*, which provides some form of **service discovery** to the nodes that are part of the network. The nodes can query this service to discover other nodes that are part of the network in order to transact with them. Interestingly, the nodes do not fully trust the network operator for the distribution of this information, so each entry of this map that contains the identifying data of a node (i.e. IP address, port, X.500 name, public key, X.509 certificate etc.) is also signed by the corresponding node. In order to avoid censorship by the network operator, the nodes can even exchange the files that contain this information with each other out-of-band and install them locally.

Let's have a look at the data model of Corda now. The shared facts between Corda nodes are represented by *states*, which are immutable objects which can contain arbitrary data depending on the use case. Since states are immutable, they cannot be modified directly to reflect a change in the state of the world. Instead, the current state is marked as historic and is replaced by a new state, which creates a chain of states that gives us a full view of the evolution of a shared fact over time. This evolution is done by a Corda *transaction*, which specifies the states that are marked as historic (also known as the *input states* of the transaction) and the new states that supersede them (also known as the *output states* of the transaction). Of course, there are specific rules that specify what kind of states each state can be replaced by. These rules are specified by *smart contracts* and each state also contains a reference to the contract that governs its evolution. The smart contract is essentially a *pure function* that takes a transaction as an input and determines whether this transaction is considered valid based on the contract's rules. Transactions can also contain *commands*, which indicate the transaction's intent in terms of how the data of the states are used. Each command is also associated with a list of public keys that need to sign the transaction in order to be valid.

Figure 7.24 contains a very simple example of this data model for electronic money. In this case, each state represents an amount of money issued by a specific bank and owned by a specific party at some point in time. We can see that Alice combines two cash states in order to perform a payment and transfer 10 GBP to Bob. After that, Bob decides to redeem this money in order to get some cash from the bank. As shown in the diagram, there are two different commands for each case. We can also guess some of the rules of the associated contract for this cash state. For a `Spend` command, the contract will verify that the sum of all input states equals the sum of all output states, so that no money is lost or created out of thin air. Most likely, it will also check that the `Spend` command contains all the owners of the input states as signers, which need to attest to this transfer of money.

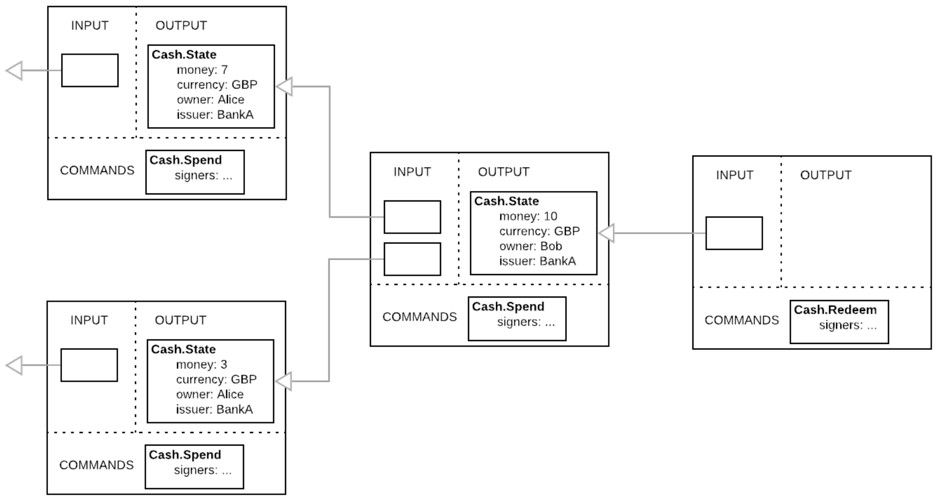


Figure 7.24: Corda data model

The astute reader will notice that in this case nothing would prevent someone from spending a specific cash state to two different parties who would not be able to detect that. This is known as *double spend* and it's prevented in Corda via the concept of *notaries*. A notary is a Corda service that is responsible for attesting that a specific state has not been spent more than once. In practice, every state is associated with a specific notary and every transaction that wants to spend this state needs to acquire a signature from this notary that proves that the state has not been spent already by another transaction. This process is known as *transaction finalisation* in Corda. The notarisation services are not necessarily provided by a single node, it can also be a *notary cluster* of multiple nodes in order to provide better **fault tolerance** and **availability**. In that case, these nodes will form a **consensus group**. Corda allows the consensus algorithm used by the notary service to be pluggable depending on the requirements in terms of privacy, scalability, performance etc. For instance, a notary cluster might choose to use a crash fault tolerant (CFT) consensus algorithm (e.g. Raft) that provides high performance but also requires high trust between the nodes of the cluster. Alternatively, it might choose to use a byzantine fault tolerant (BFT) algorithm that provides lower performance but also requires less trust between the nodes of the cluster.

At this point, it's important to note that permissioning has different implications on regular Corda nodes and notaries. In the first case, it forms the foundation for authentication of communication between nodes, while in the second case it makes it easier to detect when a notary service deviates from a protocol (e.g. violating finality), identify the associated real-world entity and take the necessary actions. This means that finalised transactions are not reversible in Corda unless someone violates the protocol[37]. As mentioned previously, in some cases even some limited amount of protocol violation can be tolerated, i.e. when using a byzantine consensus protocol.

The size of the ledger of all Corda applications deployed in a single network can become pretty large. The various nodes of the network communicate on a peer-to-peer fashion only with the nodes they need to transact, but the notary service seems to be something that needs to be used by all the nodes and could potentially end up being a **scalability** and **performance** bottleneck. For this purpose, Corda supports both **vertical** and **horizontal** partitioning. Each network can contain multiple notary clusters, so that different applications can make use of different clusters (vertical partitioning). Even the same application can choose to distribute its states between multiple notary clusters for better performance and scalability (vertical partitioning). The only requirement is for all input states of a transaction to belong to the same notary. This is so that the operation of checking whether a state is spent and marking it as spent can be done atomically in a simple and efficient way without the use of **distributed transaction** protocols. Corda provides a special transaction type, called *notary-change* transaction, which allows one to change the notary associated with a state by essentially spending the state and creating a new one associated with the new notary. However, in some use cases datasets can be partitioned in a way that requires a minimal number of such transactions, because the majority of transactions are expected to access states from the same partition. An example of this is partitioning of states according to geographic regions if we know in advance that most of the transactions will be accessing data from the same region. This architecture also makes it possible to use states from different applications in a very easy way without the use of distributed transaction protocols[38].

Corda applications are called *CorDapps* and contain several components of which the most important ones are the *states*, their *contracts* and the *flows*.

[37]This is in contrast to some other distributed ledger systems where nodes are anonymous and can thus collude in order to revert historic transactions, such as Bitcoin[74].

[38]This is known as *atomic swap* and a real use case in the financial world is known as *delivery-versus-payment* (DvP).

The flows define the workflows between nodes used to perform an update to the ledger or simply exchange some messages. Corda provides a framework that allows the application to define the interaction between nodes as a set of blocking calls that send and receive messages and the framework is responsible for transforming this to an asynchronous, event-driven execution. Corda also provides a custom serialization framework that determines how application messages are serialised when sent across the wire and how they are deserialised when received. Messaging between nodes is performed with the use of message queues, using the Apache ActiveMQ Artemis message broker. Specifically, each node maintains an inbound queue for messages received by other nodes and outbound queues for messages sent to other nodes along with a bridge process responsible for forwarding messages from the node's outbound queues to the corresponding inbound queues of the other nodes. Even though all of these moving parts can crash and restart in the middle of some operation, the platform provides the **guarantee** that every node will process each message **exactly-once**. This is achieved by resending messages until they are acknowledged and having nodes keeping track of messages processed already and discarding duplicates. Nodes also need to acknowledge a message, store its identifier and perform any related side-effects in an atomic way, which is achieved by doing all of this in a single database transaction. All the states from the ledger that are relevant to a node are stored in its database, this part of the database is called the *vault*. A node provides some more APIs that can be used for various purposes, such as starting flows or querying the node's vault. These can be accessed remotely via a client, which provides a **remote procedure call (RPC)** interface that's implemented on top of the existing messaging infrastructure and using the serialization protocol described before. Figure 7.25 contains a high-level overview of the architecture described so far.

Corda is a very interesting case study from the perspective of **backwards compatibility**. In a distributed system, the various nodes of the system might be running different versions of the software, since in many cases software has to be deployed incrementally to them and not in a single step. In a decentralised system, there is an additional challenge, because the various nodes of the systems are now controlled by different organisations, so these discrepancies might last longer. Corda provides a lot of different mechanisms to preserve backwards compability in different areas, so let's explore some of them. First of all, Corda provides API & ABI[39] backwards compatibility for all the public APIs available to CorDapps. This means that

[39]See: https://en.wikipedia.org/wiki/Application_binary_interface

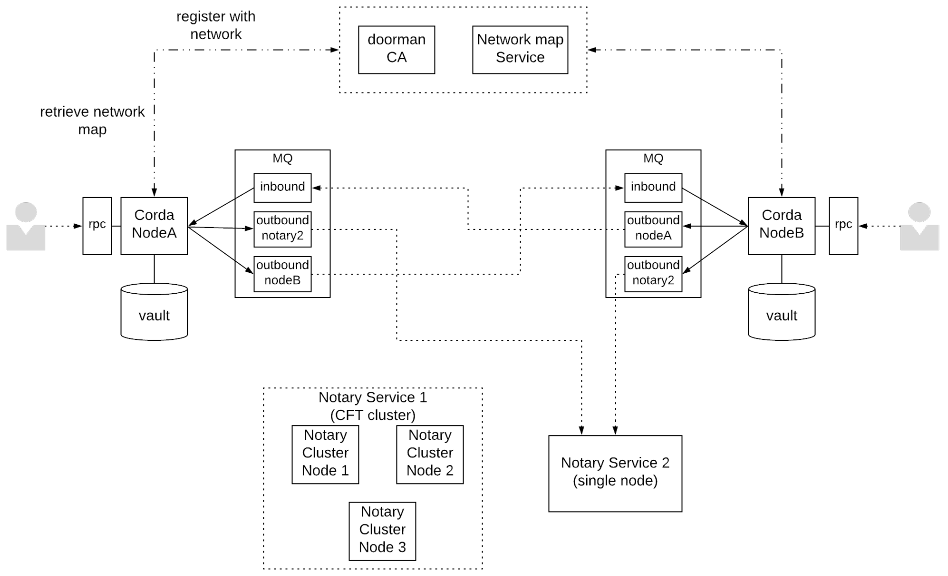


Figure 7.25: High-level overview of Corda's architecture

any CorDapp should be able to run in future versions of the platform without any change or re-compilation. Similar to other applications, CorDapps are expected to evolve, which might involve changing the structure of data exchanged between nodes and the structure of data stored in the ledger (e.g. states). The serialization framework provides some support for evolution for the first case. For instance, nullable properties can be added to a class and the framework will take care of the associated conversions. A node running an older version of the CorDapp will just ignore this new property if data is sent from a node running a newer version of the CorDapp. A node running a newer version of the CorDapp will populate the property with null when receiving data from a node running the older version of the CorDapp. Removing nullable properties and adding a non-nullable property is also possible by providing a default value. However, the serialization framework does not allow this form of data loss to happen for data that are persisted in the ledger, such as states and commands. Since states can evolve and the ledger might contain states from many earlier versions of a CorDapp, newer versions of a contract need to contain appropriate logic that is able to process states from earlier versions of the CorDapp. The contract logic for handling states from version v_i of the CorDapp can be removed by a subsequent release of the CorDapp only after all unspent states in the ledger are from version v_j of the CorDapp, where $j > i$.

In some cases, the platform might introduce a new feature that is not backwards compatible, i.e. cannot be understood by older versions of the platform. This can be problematic for two reasons. First of all, two nodes running different versions of the platform might reach different conclusions with regards to the validity of a transaction. Furthermore, when validating a transaction nodes are supposed to also validate previous transactions that were involved in the chain of provenance of the states that are consumed in the current transaction. This means that a node running an older version of the platform might fail to validate a transaction that was deemed valid in the past, because it is using a feature introduced in a newer version of the platform. Corda solves this problem with the use of the *network minimum platform version*. Every network comes with a set of parameters that every node participating in the network needs to agree on and to use to correctly interoperate with each other. This set contains a parameter called `minimumPlatformVersion`, which determines the minimum platform version that the nodes must be running, any node which is below this will not be able to start. Any feature of the platform that is not backwards compatible and requires a minimum version of the platform can check this parameter and be enabled only when the network is over a specific platform version. In this way, the nodes of a network can start using a feature only after they can be certain all other nodes will also be able to use it. This establishes a balance between nodes in a network that are keen on using a new feature and nodes that are risk averse and are not willing to upgrade to a new version of the platform.

However, all of this applies only to features that have network-wide implications, e.g. ones that determine how data are stored on the ledger. There can also be features that do not affect the whole network. Examples of this are changes to the way two nodes interact during the execution of a flow or even the way a single node executes some part of a CorDapp locally. Corda provides multiple versioning levers for more fine-grained control. For this purpose, CorDapps provide two version numbers: `minimumPlatformVersion` and `targetPlatformVersion`. The former indicates the minimum platform version the node must have for the CorDapp to function properly, which is essentially determined based on the features that are necessary to the CorDapp and the platform version they were introduced in. The latter indicates the highest version the CorDapp has been tested against and helps the platform disable backwards compatibility codepaths that might make the CorDapp less efficient or secure. Note that these versions only have implications on the node running the CorDapp, instead of the whole network. Another example is the fact that flows in a CorDapp can be versioned in order to evolve while maintaining backwards compatibility. In this way, a flow can behave differently depending on the version of the CorDapp that is deployed on the counterparty node. This makes it possible to upgrade a CorDapp incrementally across various nodes, instead of all of them having to do it in lockstep.

Distributed data processing systems

This section will examine distributed systems that are used to process large amounts of data that would be impossible or very inefficient to process using only a single machine. They can be classified in two main categories:

- **batch processing systems**: these systems group individual data items into groups, called *batches*, which are processed one at a time. In many cases, these groups can be quite large (e.g. all items for a day), so the main goal for these system is usually to provide high throughput sometimes at the cost of increased latency.
- **stream processing systems**: these systems receive and process data continuously as a *stream* of data items. As a result, the main goal for these systems is providing very low latency sometimes at the cost of decreased throughput.

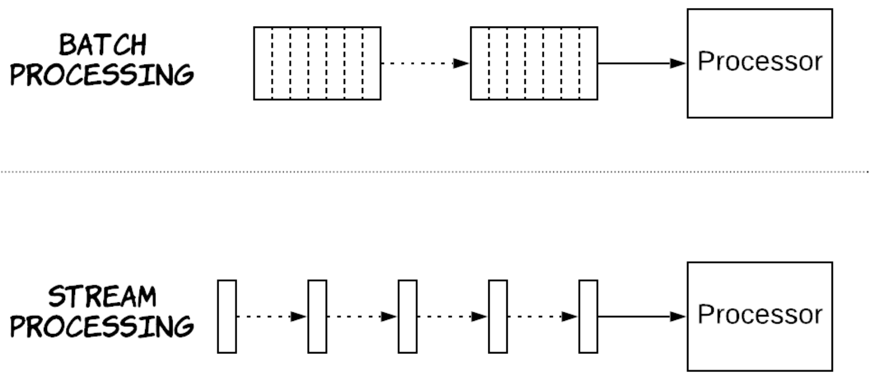


Figure 7.26: Batch and stream processing

There is also a form of processing that is essentially a hybrid between these two categories, called **micro-batch processing**. This approach processes data in batches, but these batches are kept very small in order to achieve a balance between throughput and latency.

MapReduce

MapReduce was a framework for batch data processing originally developed internally in Google[75] that was later incorporated in the wider Apache Hadoop framework. The framework draws inspiration from the field of functional programming and is based on the following main idea. Many real-word computations can be expressed with the use of two main primitive functions, *map* and *reduce*. The map function processes a set of key/value pairs and produces as output another set of intermediate key/value pairs. The reduce function receives all the values for each key and returns a single value, essentially merging all the values according to some logic. These primitive functions have a very important property, they can easily be parallelised and run across multiple machines for different parts of the dataset. As a result, the application code is responsible for defining these two methods and the framework is responsible for **partitioning the data**, scheduling the program's execution across multiple nodes, **handling node failures** and managing the required inter-machine communication.

Let's see a typical example to understand better how this programming model works in practice. We assume we have a huge collection of documents (e.g. webpages) and we need to count the number of occurrences for each word. To achieve that via MapReduce, we would use the following functions:

```
// key: the document name
// value: the document contents
map(String key, String value) {
    for(word: value.split(" ")) {
        emit(word, 1)
    }
}


reduce(String key, Iterator<Integer> values) {
    int count = 0;
    for(value: values) {
        result += value;
    }
    emit(key, result);
}
```

In this case, the map function would emit a single record for each word with the value 1, while the reduce function would just count all these entries and return the final sum for each word.

This framework is also based on a master-worker architecture, where the master node is responsible for scheduling tasks at worker nodes and managing their execution, as shown in Figure 7.27. Apart from the definition of the map/reduce functions, the user can also specify the number `M` of map tasks, the number `R` of reduce tasks, the input/output files and a partitioning function that defines how key/value pairs from the map tasks are partitioned before being processed by the reduce tasks. By default, a hash partitioner is used that selects a reduce task using the formula `hash(key) mod R`. The execution of a MapReduce proceeds in the following way:

- The framework divides the input files into `M` pieces, called *input splits*, which are typically between 16 and 64 MB per split.
- It then starts an instance of a master node and multiple instances of worker nodes on an existing cluster of machines.
- The master selects idle worker nodes and assigns map tasks to them.
- A worker node that is assigned a map task reads the contents of the associated input split, it parses key/value pairs and passes them to the user-defined map function. The entries emitted by the map function are buffered in memory and periodically written to the local disk, partitioned into `R` regions using the partitioning function. When a worker node completes a map task, it sends the location of the local file to the master node.
- The master node assigns reduce tasks to worker nodes providing the location to the associated files. These worker nodes then perform remote procedure calls (RPCs) to read the data from the local disks of the map workers. The data is first sorted[40], so that all occurrences of the same key are grouped together and then passed into the reduce function.
- When all map and reduce tasks are completed, the master node returns the control to the user program. After successful completion, the output of the mapreduce job is available in the `R` output files that can either be merged or passed as input to a separate MapReduce job.

[40] If the size is prohibitively large to fit in memory, external sorting is used.

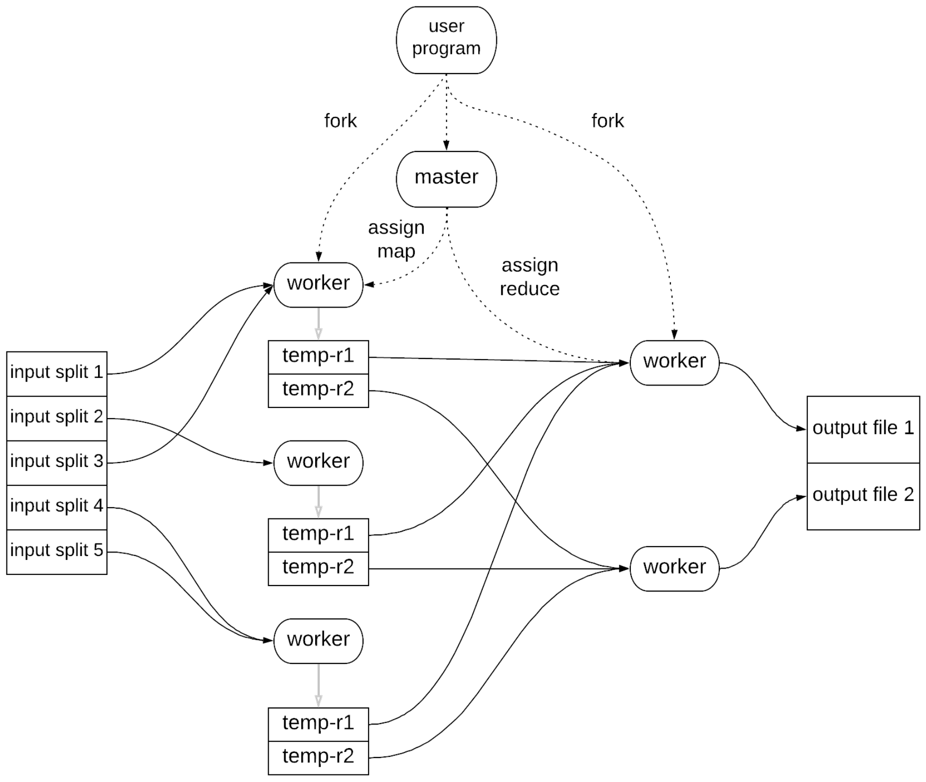


Figure 7.27: Architecture of MapReduce

The master node communicates with every worker periodically in the background as a form of **heartbeat**. If no response is received for a specific amount of time, the master node considers the worker node as failed and re-schedules all its tasks for re-execution. More specifically, reduce tasks that had been completed do not need to be rerun, since the output files are stored in an external file system. However, map tasks are rerun regardless of whether they had completed, since their output is stored on the local disk and is therefore inaccessible to the reduce tasks that need it.

This means that **network partitions** between the master node and worker nodes might lead to multiple executions of a single map or reduce task. Duplicate executions of map tasks are **deduplicated** at the master node, which ignores completion messages for already completed map tasks. Reduce tasks write their output to a temporary file, which is atomically renamed when the reduce task completes. This atomic rename operation provided by the underlying file system guarantees that the output files will contain just the data produced by a single execution of each reduce task. However, if the map/reduce functions defined by the application code have additional side-effects (e.g. writing to external datastores) the framework does not provide any guarantees and the application writer needs to make sure these **side-effects** are **atomic** and **idempotent**, since the framework might trigger them more than once as part of a task re-execution.

Input and output files are usually stored in a distributed filesystem, such as HDFS or GFS. MapReduce can take advantage of this to perform several optimisations, such as scheduling map tasks on worker nodes that contain a replica of the corresponding input to minimize network traffic or aligning the size of input splits to the block size of the file system.

The framework provides the guarantee that within a given partition, the intermediate key/value pairs are processed in increasing key order. This ordering guarantees makes it easy to produce a sorted output file per partition, which is useful for use cases that need to support efficient random access lookups by key or need sorted data in general. Furthermore, some use-cases would benefit from some form of pre-aggregation at the map level to reduce the amount of data transferred between map and reduce tasks. This was evident in the example presented above, where a single map would emit multiple entries for each occurence of a word, instead of a single entry with the number of occurrences. For this reason, the framework allows the application code to also provide a *combine* function. This method has the same type as the reduce function and is run as part of the map task in order to pre-aggregate the data locally.

Apache Spark

Apache Spark [76][77] is a data processing system that was initially developed at the University of California and then donated to the Apache Software Foundation. It was developed in response to some of the limitations of MapReduce. Specifically, the MapReduce model allowed developing and running embarrassingly parallel computations on a big cluster of machines, but every job had to read the input from disk and write the output to disk. As a result, there was a lower bound in the latency of a job execution, which was determined by disk speeds. This means MapReduce was not a good fit for iterative computations, where a single job was executed multiple times or data were passed through multiple jobs, and for interactive data analysis, where a user wants to run multiple ad-hoc queries on the same dataset. Spark tried to address these two use-cases.

Spark is based on the concept of *Resilient Distributed Datasets* (RDD), which is a distributed memory abstraction used to perform in-memory computations on large clusters of machines in a fault-tolerant way. More concretely, an RDD is a **read-only**, **partitioned** collection of records. RDDs can be created through operations on data in stable storage or other RDDS. The operations performed on an RDD can be one of the following two types:

- *transformations*, which are lazy operations that define a new RDD. Some examples of transformations are `map`, `filter`, `join` and `union`.
- *actions*, which trigger a computation to return a value to the program or write data to external storage. Some examples of actions are `count`, `collect`, `reduce` and `save`.

A typical Spark application will create an RDD by reading some data from a distributed filesystem, it will then process the data by calculating new RDDs through transformations and will finally store the results in an output file. For example, an application used to read some log files from HDFS and count the number of lines that contain the word "sale completed" would look like the following:

```
lines = spark.textFile("hdfs://...")
completed_sales = lines.filter(_.contains("sale completed"))
number_of_sales = completed_sales.count()
```

This program can either be submitted to be run as an invividual application in the background or each one of the commands can be executed interactively in the Spark interpreter. A Spark program is executed from a coordinator process, called the *driver*. The Spark cluster contains a *cluster manager* node and a set of *worker nodes*. The responsibilities between these components are split in the following way:

- The cluster manager is responsible for managing the resources of the cluster (i.e. the worker nodes) and allocating resources to clients that need to run applications.
- The worker nodes are the nodes of the cluster waiting to receive applications/jobs to execute.
- Spark also contains a *master* process that requests resources in the cluster and makes them available to the driver[41].

[41]Note that Spark supports both a standalone clustering mode and some clustering modes

- The driver is responsible for requesting the required resources from the master and starting a Spark agent process on each node that runs for the entire lifecycle of the application, called *executor*. The driver then analyses the user's application code into a directed acyclic graph (DAG) of stages, partitions the associated RDDs and assigns the corresponding tasks to the executors available to compute them. The driver is also responsible for managing the overall execution of the application, e.g. receiving heartbeats from executors and restarting failed tasks.

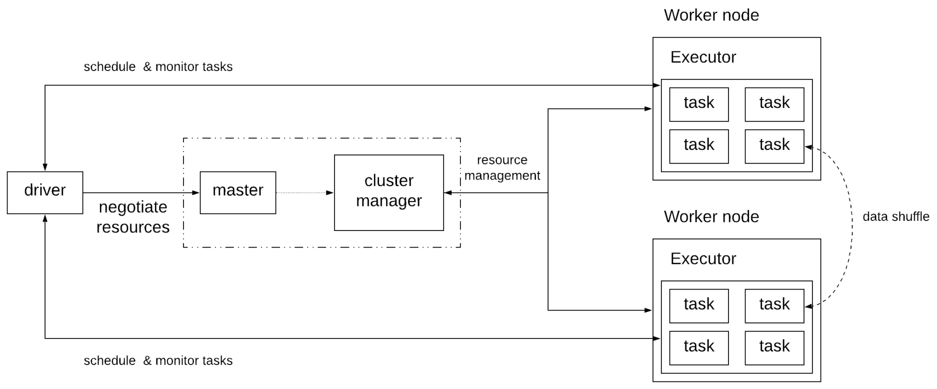


Figure 7.28: Architecture of Spark

Notably, in the previous example the second line is executed without any data being read or processed yet, since `filter` is a transformation. The data is being read from HDFS, filtered and then counted, when the third line is processed, which contains the `count` operation which is an action. To achieve that, the driver maintains the relationship between the various RDDs through a *lineage* graph, triggering calculation of an RDD and all its ancestors only when an action is performed. RDDs provide the following basic operations[42]:

- `partitions()`, which returns a list of partition objects. For example,

using third-party cluster management systems, such as YARN, Mesos and Kubernetes. In the standalone mode, the master process also performs the functions of the cluster manager. In some of the other clustering modes, such as Mesos and YARN, they are separate processes.

[42]Note that these operations are mainly used by the framework to orchestrate the execution of Spark applications. The applications are not supposed to make use of these operations, they should be using the transformations and actions that were presented previously.

an RDD representing an HDFS file has a partition for each block of the file by default. The user can specify a custom number of partitions for an RDD, if needed.

- `partitioner()`, which returns metadata determining whether the RDD is **hash/range partitioned**. This is relevant to transformations that join multiple key-value RDDs based on their keys, such as `join` or `groupByKey`. In these cases, hash partitioning is used by default on the keys, but the application can specify a custom range partitioning to be used, if needed. An example is provided in the paper [77], where execution of the PageRank algorithm on Spark can be optimised by providing a custom partitioner that groups all the URLs of a single domain in the same partition.
- `preferredLocations(p)`, which lists nodes where partition `p` can be accessed faster due to data locality. This might return nodes that contain the blocks of an HDFS file corresponding to that partition or a node that already contains in memory a partition that needs to be processed.
- `dependencies()`, which returns a list of dependencies on parent RDDs. These dependencies can be classified into two major types: *narrow* and *wide* dependencies. A narrow dependency is one where a partition of the parent RDD is used by at most one partition of the child RDD, such as `map`, `filter` or `union`. A wide dependency is one where multiple child partitions may depend on a single parent partition, such as a `join` or `groupByKey`. Note that a `join` of two RDDs can lead to two narrow dependencies, if both of the RDDs are partitioned with the same partitioner, as shown in Figure 7.29.
- `iterator(p, parentIters)`, which computes the elements of a partition `p` given iterators for its parent partitions.

As explained before, the driver examines the lineage graph of the application code and builds a DAG of *stages* to execute, where each stage contains as many pipelined transformations with narrow dependencies as possible. The boundaries of each stage correspond to operations with wide dependencies that require data shuffling between partitions or any already computed partitions that have been persisted and can short-circuit the computation of ancestor RDDs. The driver launches tasks to compute missing partitions from each stage until it has computed the target RDD. The tasks are assigned to executors based on data locality. If a task needs to process a partition that is available in memory on a node, it's submitted to that node. If a task processes a partition for which the containing RDD provides preferred

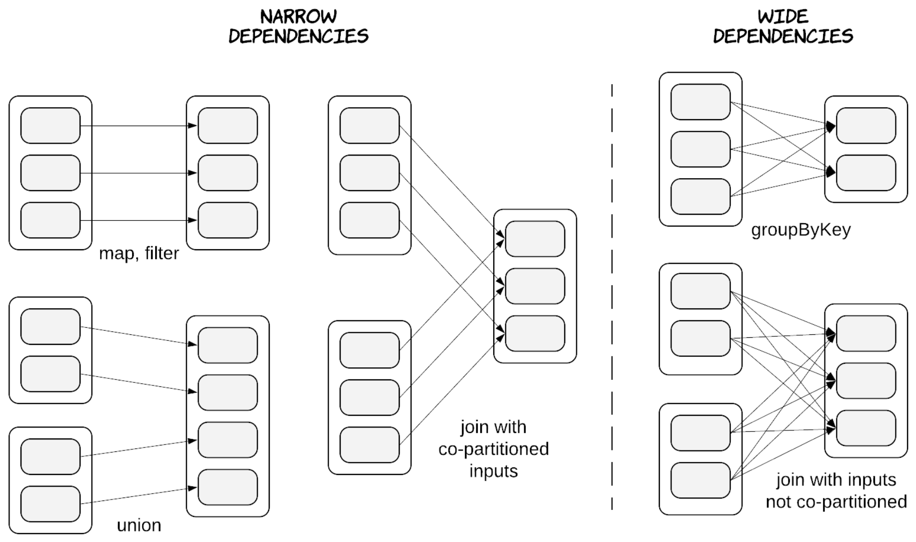

NARROW DEPENDENCIES
WIDE DEPENDENCIES
map, filter
union
join with co-partitioned inputs
groupByKey
join with inputs not co-partitioned

Figure 7.29: Examples of narrow and wide dependencies. Each big rectangle is an RDD with the smaller grey rectangles representing partitions of the RDD

locations (e.g. an HDFS file), it's submitted to these nodes. For wide dependencies that require data shuffling, nodes holding parent partitions materialize intermediate records locally that are later pulled by nodes from the next stage, similar to MapReduce.

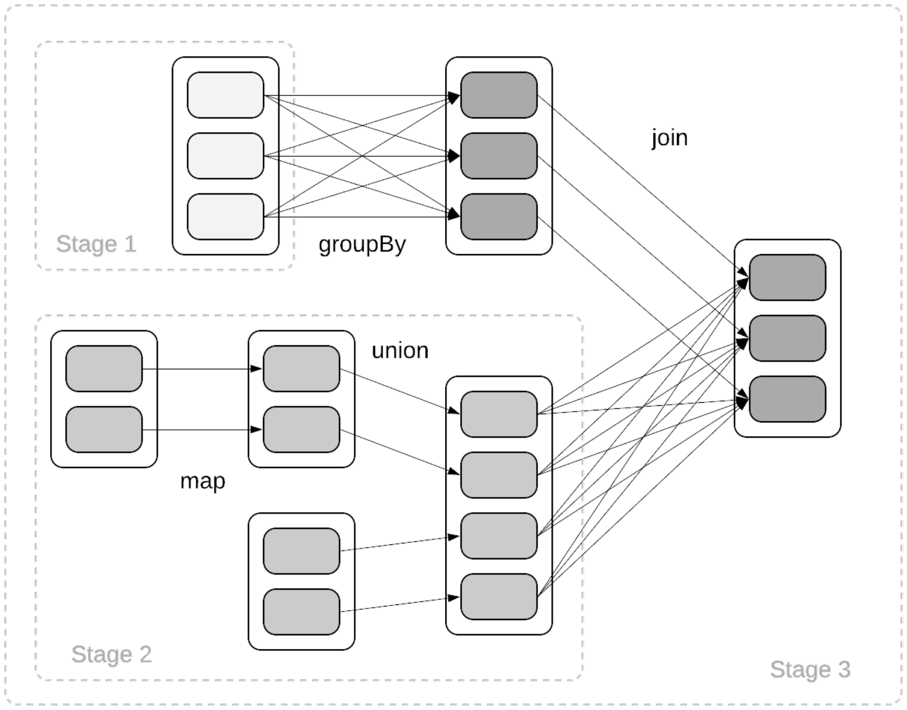


Figure 7.30: Example of a DAG of stages computed from a Spark application

This graph is the basic building block for **efficient fault-tolerance**. When an executor fails for some reason, any tasks running on it are re-scheduled on another executor. Along with this, tasks are scheduled for any parent RDDs required to calculate the RDD of this task. As a consequence of this, wide dependencies can be much more inefficient than narrow dependencies when recovering from failures, as shown in Figure 7.31. Long lineage graphs can also make recovery very slow, since many RDDs will need to be recomputed in a potential failure near the end of the graph. For this reason, Spark provides a **checkpointing** capability, which can be used to store RDDs from specified tasks to stable storage (e.g. a distributed filesystem). In this way, RDDs that have been checkpointed can be read from stable storage during recovery, thus only having to rerun smaller portions of the graph. Users can call a `persist()` method to indicate which RDDs need to be stored on disk.

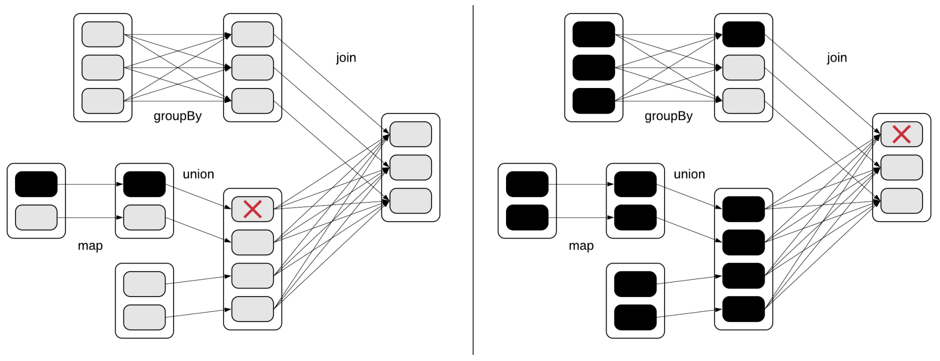


Figure 7.31: The impact of wide dependencies on recovery from failures. The red cross indicates failure of a task calculating a specific partition of an RDD. The black rectangles represent the partitions of RDDs that need to be recomputed, if not persisted.

The `persist()` method provides different storage levels depending on the needs of the application. As explained before, persistence on disk can be used to make recovery from failures faster. Users can also instruct nodes to store calculated RDDs in memory, so that they can be served from memory every time they are needed and they don't need to be recalculated. This option is very useful for interactive applications, where a specific RDD is calculated and is then used in multiple different ways in order to explore a dataset without having to calculate the whole lineage each time. Spark also provides graceful degradation in cases where memory is not enough, so that the application does not fail but keeps running with decreased performance. For instance, Spark can either recalculate any partitions on demand when they don't fit in memory or spill them to disk. Wide dependencies cause more data to be exchanged between nodes compared to narrow dependencies, so performance can be increased significantly by reducing wide dependencies or the amount of data that need to be shuffled. One way to do this is by pre-aggregating data, also known as *map-side reduction*[43]. As an example, the following code performs a word count in two different ways: the first one will send multiple records of value 1 for each word across the network, while

[43] As explained previously, this is a capability provided in the MapReduce framework too through the use of *combiners*.

the second one will send one record for each word containing the number of occurrences.

```
// word count without pre-aggregation
sparkContext.textFile("hdfs://...")
    .flatMap(line => line.split(" "))
    .map(word => (word,1))
    .groupByKey()
    .map((x,y) => (x,sum(y)))

// word count with pre-aggregation
sparkContext.textFile("hdfs://...")
    .flatMap(line => line.split(" "))
    .map(word => (word,1))
    .reduceByKey((x,y)=> (x+y))
```

Spark can also be configured in a way that is resilient to failures of the master process. This is achieved via Zookeeper, where all the masters are performing leader election and one of them is elected as the leader with the rest remaining in standby mode. When a new worker node is added to the cluster, it registers with the master node. If failover occurs, the new master will contact all previously registered worker nodes to inform them of the change in leadership. So, only the scheduling of new applications is affected during a failover, applications that were already running are unaffected. When submitting a job to the Spark cluster, the user can specify through a `--supervise` option that the driver needs to be automatically restarted by the master if it fails with non-zero exit code.

Spark supports multiple, different systems for data persistence. It has a commit protocol that aims to provide **exactly-once guarantees** on the job's output under specific conditions. This means that no matter how many times worker nodes fail and tasks are rerun, if a job completes, there will be no duplicate or corrupt data in the final output file. This might not be the case for every supported storage system and it's achieved differently depending on the available capabilities. For instance, when using HDFS, each task writes the output data to a unique, temporary file (e.g. `targetDirectory/_temp/part-XXXX_attempt-YYYY`) and when the write is complete, the file is moved to the final location (e.g. `targetDirectory/part-XXXX`) using an **atomic** rename operation provided by HDFS. As a result, even if a task is executed multiple times due to failures, the final file will contain its output exactly once. Furthermore, no matter which execution completed successfully, the output data will be the same as long as the transformations that were used were **deterministic** and **idempotent**. This is true for any transformations that act solely on the data provided by the previous RDDs using deterministic operations. However, this is not the case if these transformations make use of data from other systems that might change between executions or if they use non-deterministic actions (e.g. mathematical calculations based on random number generation). Furthermore, if transformations perform side-effects on external systems that are not idempotent, no guarantee is provided since Spark might execute each side-effect more than once.

Apache Flink

Apache Flink[78][79] is an open-source stream-processing framework developed by the Apache Software Foundation aiming to provide a high-throughput, low latency data processing engine[44].

The basic constructs in Flink are *streams* and *transformations*. A stream is an unbounded[45] flow of data records and a transformation is an operation that takes one or more streams as input and produces one or more output streams as a result. Flink provides many different APIs that can be used to define these transformations. For example, the `ProcessFunction` API is a low-level interface that allows the user to define imperatively what each transformation should do by providing the code that will process each record. The `DataStream` API is a high-level interface that allows the user to define declaratively the various transformations by re-using basic operations, such as `map`, `flatMap`, `filter`, `union` etc. Since streams can be unbounded, the application has to produce some intermediate, periodic results. For this purpose, Flink provides some additional constructs, such as *windows*, *timers* and local storage for *stateful* operators. Flink provides a set of high-level operators that specify the windows over which data from the stream will

[44]This is a main differentiator between Flink and Spark. Flink processes incoming data as they arrive, thus managing to provide sub-second latency that can go down to single-digit millisecond latency. Note that Spark also provides a streaming engine, called Spark Streaming[80]. However, that is running some form of micro-batch processing, where an input data stream is split into batches, which are then processed to generate the final results in batches with the associated latency trade-off.

[45]Note that Flink can also execute batch processing applications, where data is treated as a bounded stream. As a result, the mechanisms described in this section are used in this case too with slight variations.

be aggregated. These windows can be time-driven (e.g. time intervals) or data-driven (e.g. number of elements). The timer API allows applications to register callbacks to be executed at specific points in time in the future.

A data processing application in Flink can be represented as a directed acyclic graph (DAG)[46], where nodes represent computing tasks and edges represent data subscriptions between tasks. Flink is responsible for translating the logical graph corresponding to the application code to the actual physical graph that will be executed. This includes logical dataflow optimisations, such as fusion of multiple operations to a single task (e.g. combination of two consecutive filters). It also includes partitioning each task into multiple instances that can be executed in parallel in different compute nodes. Figure 7.32 illustrates this process.

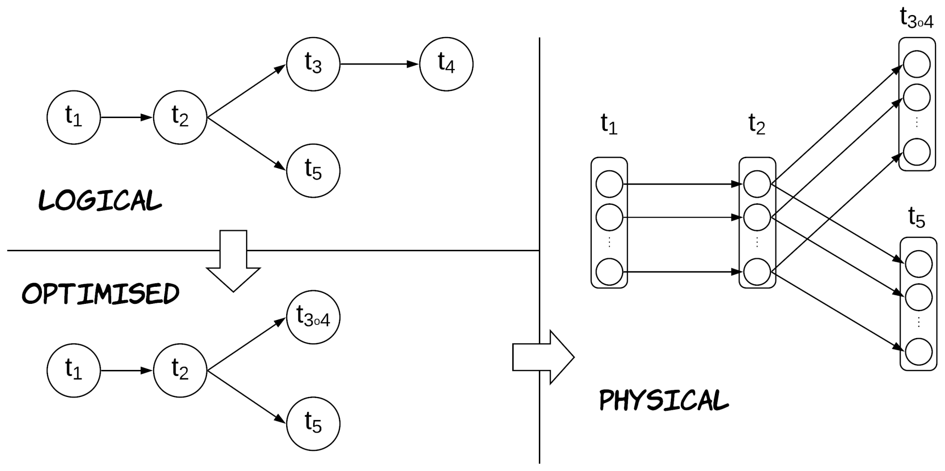


Figure 7.32: A Flink dataflow graph example

The high-level architecture of Flink consists of 3 main components, as shown in Figure 7.33. The client receives the program code, transforms it into a dataflow graph and submits it to the Job Manager, which coordinates the distributed execution of the dataflow. The Job Manager is responsible for scheduling execution of tasks on Task Managers, tracking their progress and coordinating checkpoints and recovery from possible failures of tasks. Each Task Manager is responsible for executing one or more tasks that

[46]Note that Flink also has support for cyclic dataflow graphs, which can be used for use-cases such as iterative algorithms. However, this detail is omitted here for the sake of simplicity.

execute user-specified operators that can produce other streams and report their status to the Job Manager along with heartbeats that are used for **failure detection**. When processing unbounded streams, these tasks are supposed to be long-lived. If they fail, the Job Manager is responsible for restarting them. To avoid making the Job Manager a single point of failure, multiple instances can be running in parallel. One of them will be elected leader via Zookeeper and will be responsible for coordinating the execution of applications, while the rest will be waiting to take over in case of a leader failure. As a result, the leader Job Manager stores some critical metadata about every application in Zookeeper, so that it's accessible to newly elected leaders.

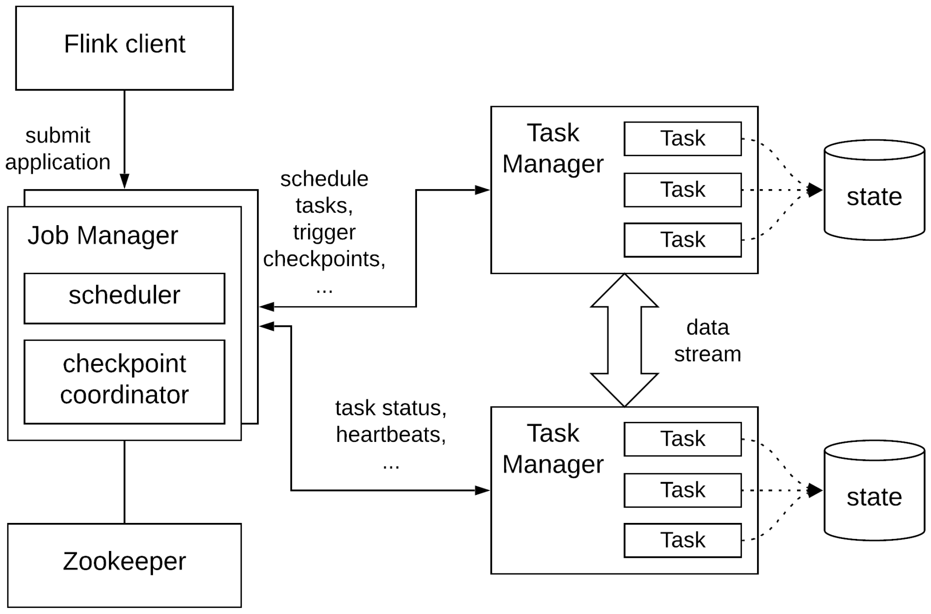


Figure 7.33: Apache Flink architecture

As explained previously, time is a crucial element that is commonly used to define boundaries on unbounded streams. Similar to other stream processing systems[81][82], Flink supports two main **notions of time**: *processing time* and *event time*[47]. Processing time refers to the system time of the machine that is executing an operation. Event time is the time that each individual

[47]In fact, Flink has a third notion of time, called the *ingestion time*. This corresponds to the time an event enters Flink. The discussion in this section will focus on event and processing time.

event occured on its producing device. Each one of them can be used with some trade-offs:

- When a streaming program runs on processing time, all time-based operations (e.g. time windows) will use the system clock of the machines that run the respective operation. This is the simplest notion of time and requires no coordination between the nodes of the system. It also provides good performance and reliably low latency on the produced results. However, all this comes at the cost of consistency and non-determinism. The system clocks of different machines will differ and the various nodes of the system will process data at different rates. As a consequence, different nodes might assign the same event to different windows depending on timing.
- When a streaming program runs on event time, all time-based operations will use the event time embedded within the stream records to track progress of time, instead of system clocks. This brings consistency and determinism to the execution of the program, since nodes will now have a common mechanism to track progress of time and assign events to windows. However, it requires some coordination between the various nodes, as we will see below. It also introduces some additional latency, since nodes might have to wait for out-of-order or late events.

The main mechanism to track progress in event time in Flink is *watermarks*. Watermarks are control records that flow as part of a data stream and carry a timestamp `t`. A `Watermark(t)` record indicates that event time has reached time `t` in that stream, which means there should be no more elements with a timestamp `t'` $\leq$ `t`. Once a watermark reaches an operator, the operator can advance its internal event time clock to the value of the watermark. Watermarks can be generated either directly in the data stream source or by a watermark generator in the beginning of a Flink pipeline. The operators later in the pipeline are supposed to use the watermarks for their processing (e.g. to trigger calculation of windows) and then emit them downstream to the next operators. There are many different strategies for generating watermarks. An example is the `BoundedOutOfOrdernessGenerator`, which assumes that the latest elements for a certain timestamp `t` will arrive at most `n` milliseconds after the earliest elements for timestamp `t`. Of course, there could be elements that do not satisfy this condition and arrive after the associated watermark has been emitted and the corresponding windows have been calculated. These are called *late* elements and Flink provides different ways to deal with them, such as discarding them or re-triggering the calculation of the associated window. Figure 7.34 contains an illustration of how watermarks flow in a Flink pipeline and how event time progresses.

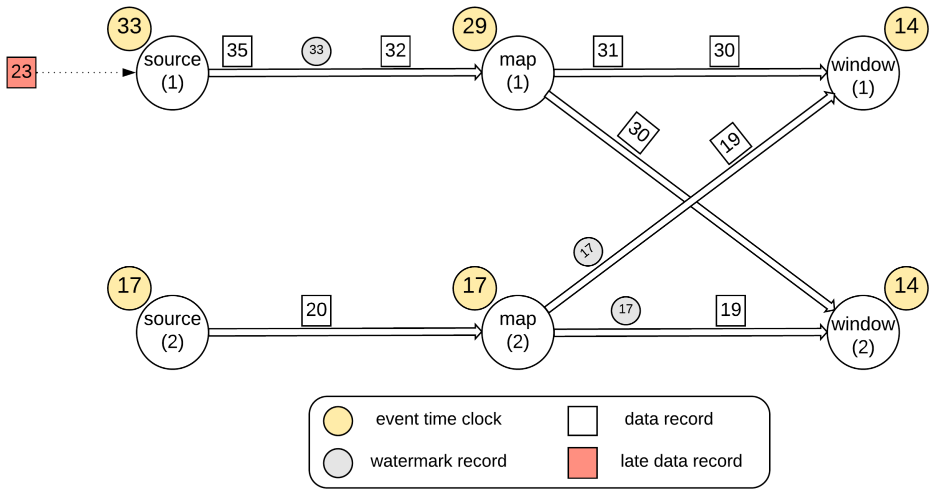


Figure 7.34: Event time & watermarks in Flink

As mentioned previously, stream processing applications in Flink are supposed to be long-lived. As a result, there must be an efficient way to recover from failures without having to repeat a lot of work. For this purpose, Flink periodically checkpoints the operators' state along with the position of the stream that has been consumed to generate this state. In case of a failure, an application can be restarted from the latest checkpoint and continue processing from there. All this is achieved via an algorithm similar to the Chandy-Lamport algorithm for distributed snapshots, called Asynchronous Barrier Snapshotting (ABS)[83]. This algorithm operates slightly different for acyclic and cyclic graphs, so we will examine the first case here which is a bit simpler. The algorithms works in the following way:

- The Job Manager periodically injects some control records in the stream, referred to as *stage barriers*. These records are supposed to divide the stream into stages, where the set of operator states at the end of a stage reflects the whole execution history up to the associated barrier and thus it can be used for a snapshot.
- When a source task receives a barrier, it takes a snapshot of its current state and it then broadcasts the barrier to all its outputs.
- When a non-source task receives a barrier from one of its inputs, it blocks that input until it has received a barrier from all the inputs. It then takes a snapshot of its current state and broadcasts the bar-

rier to its outputs. Finally, it unblocks its inputs. This blocking is needed, so that the checkpoint is guaranteed to contain the state after processing all the elements before the barrier and no elements after the barrier. Note that the snapshot taken while the inputs are blocked can be a logical one, where the actual, physical snapshot is happening asynchronously in the background[48]. This is done in order to reduce the duration of this blocking phase, so that the application can start processing data again as quickly as possible.
- Once the background copy process has completed, each task acknowledges the checkpoint back to the Job Manager. After the Job Manager has received an acknowledgement from all the tasks, the checkpoint is considered complete and can be used for recovery if a failure happens later on. At this point, the Job Manager notifies all the tasks that the checkpoint is complete, so that they can perform any cleanup or bookkeeping logic required.

There are 2 subtle points in the checkpoint algorithm and the recovery process. During recovery, tasks will be reset back to the last checkpoint and start processing again from the first element after the checkpoint was taken. This implies that any state that might have been produced by elements after the last checkpoint will be essentially discarded, so that each element is **processed exactly-once**. However, this raises the following questions:

- How is the state produced after the last checkpoint discarded in practice if it's been persisted in the operator's state?
- What happens with sink tasks that interact with external systems and records that might have been emitted after the last checkpoint in case of a failure?
- What happens with sources that do not support replay of past records?

The answers to all these questions partially rely on a core characteristic of the checkpoint algorithm. As explained previously, the algorithm has the form of a **two-phase commit protocol**, where the first phase consists of the Job Manager instructing all the tasks to create a checkpoint and the second phase consists of the Job Manager informing them that all the tasks succcesfully managed to create a checkpoint. The state of an operator can be stored in different ways, such as in the operator's memory, in an embedded key-value store or in an external datastore. If that datastore supports MVCC, then all the updates to the state can simply be stored under a version that corresponds

[48]One way to achieve this is through copy-on-write techniques. See: https://en.wikipedia.org/wiki/Copy-on-write

to the next checkpoint. During recovery, updates that had been performed after the last checkpoint are automatically ignored, since reads will return the version corresponding to the last checkpoint. If the datastore does not support MVCC, all the state changes can be maintained temporarily in local storage in the form of a write-ahead-log (WAL), which will be committed to the datastore during the second phase of the checkpoint protocol. Flink can also integrate with various other systems that can be used to retrieve input data from (sources) or send output data to (sinks), such as Kafka, RabbitMQ etc. Each one of them provides different capabilities. For example, Kafka provides an offset-based interface, which makes it very easy to replay data records in case of recovery from a failure. A sink task just has to keep track of the offset of each checkpoint and start reading from that offset during recovery. However, message queues do not provide this interface, so Flink has to use alternative methods to provide the same guarantees. For instance, in the case of RabbitMQ, messages are acknowledged and removed from the queue only after the associated checkpoint is complete, again during the second phase of the protocol. Similarly, a sink needs to coordinate with the checkpoint protocol in order to be able to provide exactly-once guarantees. Kafka is a system that can support this through the use of its transactional client. When a checkpoint is created by the sink, the `flush()` operation is called as part of the checkpoint. After the notification that the checkpoint has been completed in all operators is received from the Job Manager, the sink calls Kafka's `commitTransaction` method. Flink provides an abstract class called `TwoPhaseCommitSinkFunction` that provides the basic methods that need to be implemented by a sink that wants to provide these guarantees (i.e. `beginTransaction`, `preCommit`, `commit`, `abort`).

To sum up, some of the guarantees provided by Flink are the following:

- Depending on the types of sources and sinks used, Flink provides exactly-once processing semantics even across failures.
- The user can also optionally downgrade to at-least-once processing semantics, which can provide increased performance.
- It's important to note that the exactly-once guarantees apply to stream records and local state produced using the Flink APIs. If an operator performs additional side-effects on systems external to Flink, then no guarantees are provided for them.
- Flink does not provide ordering guarantees after any form of repartitioning or broadcasting and the responsibility of dealing with out-of-order records is left to the operator implementation.

Chapter 8

Practices & Patterns

This chapter covers common practices and patterns used when building and operating distributed systems. These are not supposed to be exact prescriptions, but they can help you identify some of the basic approaches available to you and the associated trade-offs. It goes without saying that there are so many practices and patterns that we would never be able to cover them all. As a result, the goal is to cover some of the most fundamental and valuable ones.

Communication patterns

One of the main differentiating characteristics of distributed systems is the fact that the various nodes need to exchange data across the network boundary. In this section, we will examine how that can be achieved and what are the trade-offs of each approach. First of all, every node needs a way to transform data that reside in memory into a format that can be transmitted over the network and it also needs a way to translate data received from the network back to the appropriate in-memory representation. These processes are called **serialisation** and **deserialisation** respectively[1].

The various nodes of the system need to agree on a common way to serialise and deserialise data. Otherwise, they will not be able to translate data they

[1]Note that serialisation can also be used to transform data in a format that's suitable for storage, not only communication.

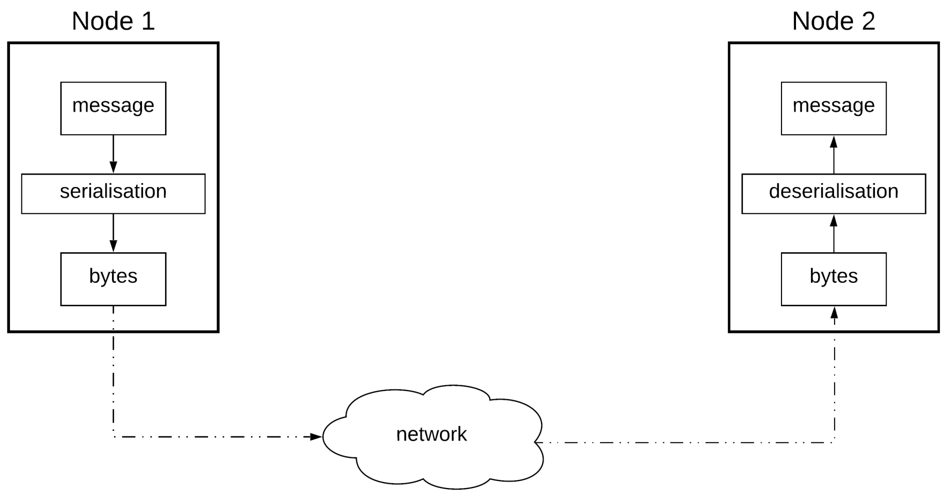


Figure 8.1: Serialisation and deserialisation

send to each other. There are various options available for this purpose, so let's have a look at some of them:

- Some languages provide native support for serialisation, such as Java and Python via its `pickle` module. The main benefit of this option is convenience, since there is very little extra code needed to serialise and deserialise an object. However, this comes at the cost of maintainability, security and interoperability. Given the transparent nature of how these serialisation methods work, it becomes hard to keep the format stable, since it can be affected even by small changes to an object that do not affect the data contained in it (e.g. implementing a new interface). Furthermore, some of these mechanisms are not very secure, since they indirectly allow a remote sender of data to initialise any objects they want, thus introducing the risk of remote code execution. Last but not least, most of these methods are available only in specific languages, which means systems developed in different programming languages will not be able to communicate. Note that there are some third-party libraries that operate in a similar way using reflection or bytecode manipulation, such as Kryo[2]. These libraries tend to be subject to the same trade-offs.
- Another option is a set of libraries that serialise an in-memory object

[2]See: https://github.com/EsotericSoftware/kryo

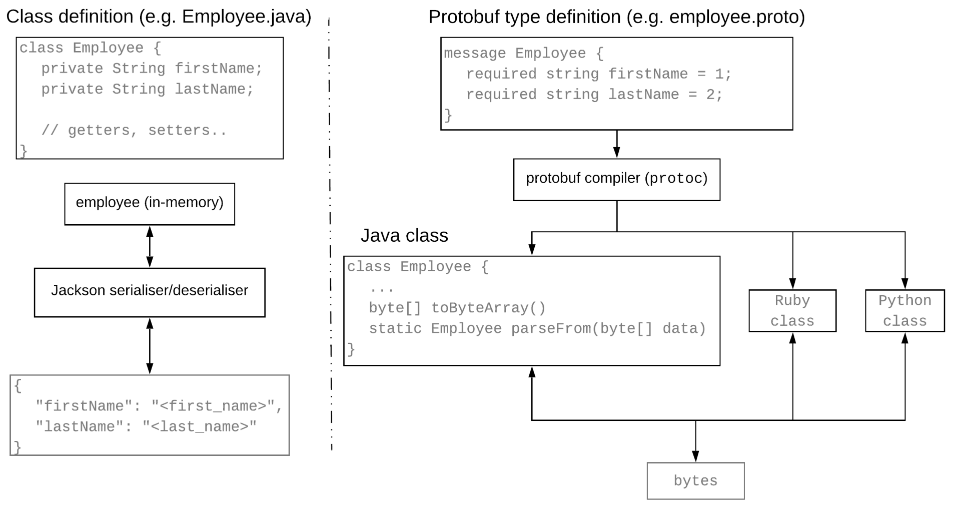
based on instructions provided by the application. These instructions can be either imperative or declarative, i.e. annotating the fields to be serialised instead of explicitly calling operations for every field. An example of such a library is Jackson[3], which supports a lot of different formats, such as JSON and XML. A main benefit of this approach is the ability to interoperate between different languages. Most of these libraries also have rather simple rules on what gets serialised, so it's easier to preserve backwards compatibility when evolving some data, i.e. introducing new optional fields. They also tend to be a bit more secure, since they reduce the surface of exploitation by reducing the number of types that can be instantiated during deserialisation, i.e. only the ones that have been annotated for serialisation and thus implicitly whitelisted. However, sometimes they can create additional development effort, since the same serialisation mapping needs to be defined on every application.

- The last option we will explore is **interface definition languages** (IDL). These are specification languages used to define the schema of a data type in a language-independent way. Typically, these definitions can then be used to dynamically generate code in different languages that will be able to perform serialisation and deserialisation, when included in the application. This can allow applications built in different programming languages to interoperate, depending on the languages supported by the IDL. Each IDL allows for different forms of evolution of the underlying data types. They also reduce duplicate development effort, but they require adjusting build processes so that they are able to integrate with the code generation mechanisms. Some examples of IDLs are Protocol Buffers[4], Avro[5] and Thrift[6].

Figure 8.2 provides an illustration of the difference between the second and the third approach, using Jackson and Protocol Buffers as examples.

So far, we have seen what are the options for creating and parsing the data sent through the network. A question that remains unanswered is what are the main ways through which this data is sent and received and the associated trade-offs. This does not refer to the underlying transfer protocols that are used, such as Ethernet, IP, TCP, UDP etc. These are actually essential to this data transfer, but they are beyond the scope of this book. Instead, it

[3]See: https://github.com/FasterXML/jackson
[4]See: https://github.com/protocolbuffers/protobuf
[5]See: https://avro.apache.org/
[6]See: https://thrift.apache.org/


Class definition (e.g. Employee.java)
employee (in-memory)
Jackson serialiser/deserialiser
Protobuf type definition (e.g. employee.proto)
protobuf compiler (protoc)
Java class
Ruby class
Python class
bytes

```
class Employee {
  private String firstName;
  private String lastName;

  // getters, setters..
}
```

```
{
  "firstName": "<first_name>",
  "lastName": "<last_name>"
}
```

```
message Employee {
  required string firstName = 1;
  required string lastName = 2;
}
```

```
class Employee {
  ...
  byte[] toByteArray()
  static Employee parseFrom(byte[] data)
}
```

Figure 8.2: Jackson and Protocol Buffers

refers to the two basic, high-level forms of communication, **synchronous** and **asynchronous** communication.

If node A is communicating synchronously with node B, this means that node A will be waiting until it has received a response from node B before proceeding with subsequent tasks. If node A is communicating asynchronously instead, this means that it does not have to wait until that request is complete; it can proceed and perform other tasks at the same time. A simple illustration of this difference can be seen in Figure 8.3. Synchronous communication is typically used in cases, where node A needs to retrieve some data in order to execute the next task or it needs to be sure a side-effect has been performed successfully on node B's system before proceeding. Asynchronous communication is preferred in cases, where the operation performed in node B can be executed independently without blocking other operations from making progress. For instance, an e-commerce system might need to know that a payment has been performed successfully before dispatching an item, thus using synchronous communication. However, after the order has been dispatched, the process that sends the appropriate notification e-mail to the customer can be triggered asynchronously.

Synchronous communication is usually implemented on top of the existing protocols, such as using HTTP in order to transmit data in JSON or some other serialisation format (e.g. protocol buffers). In the case of asynchronous

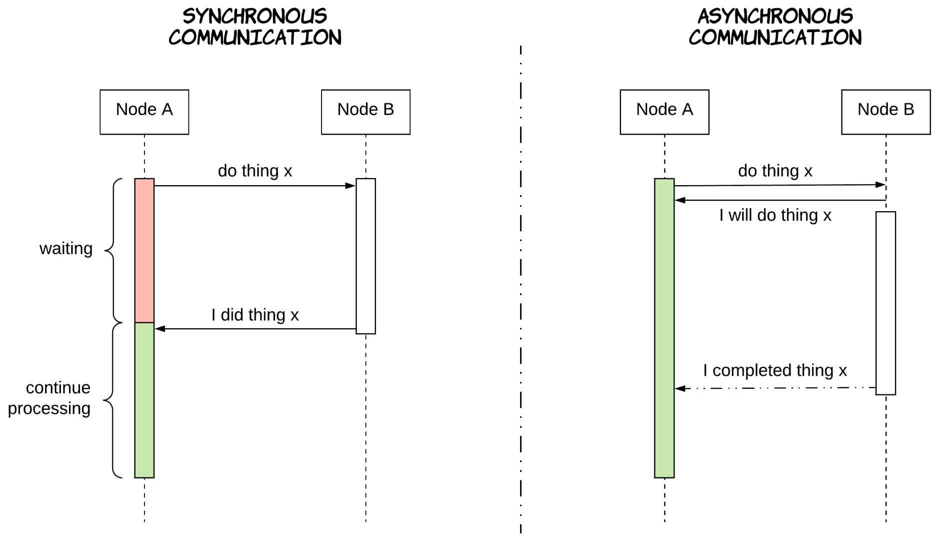


Figure 8.3: Synchronous and asynchronous communication

communication between two nodes, there is usually a need to persistently store incoming requests until they are processed. This is needed so that requests are not lost even if the recipient node fails for some reason[7]. These requests can be stored in an intermediate datastore in order to prevent loss in the face of failures. The datastores that are typically used for this purpose belong in two main categories: **message queues** and **event logs**.

Some commonly used messages queues are ActiveMQ[8], RabbitMQ[9] and Amazon SQS[10]. A message queue usually operates with first-in, first-out (FIFO) semantics. This means that messages, in general, are added to the tail of the queue and removed from the head. Note that multiple producers can be sending messages to the queue and multiple consumers can be receiving

[7]Note that this is not necessarily needed in the case of synchronous communication, because if the recipient crashes before processing a request, the sender will notice at some point and it can retry the request. But, even when using asynchronous communication, there can be cases where losing requests is not a big problem. A typical example of this is the autocomplete functionality of a search engine; this can be done asynchronously in the background and losing some requests will just lead to fewer suggestions to the user, so it might be acceptable.

[8]See: http://activemq.apache.org

[9]See: https://www.rabbitmq.com

[10]See: https://aws.amazon.com/sqs

messages and processing them concurrently. Depending on how a message queue is configured, messages can be deleted as soon as they are delivered to a consumer or only after a consumer has explicitly acknowledged it has succcessfully processed a message. The former essentially provides **at-most-once** guarantees, since a consumer might fail after receiving a message, but before acknowledging it. The latter can provide **at-least-once** semantics, since at least a single consumer must have processed a message before it being removed from the queue. However, most message queues contain a timeout logic on unacknowledged messages to cope with failed consumers ensuring *liveness*[11]. This means that unacknowledged messages are being put back to the queue and redelivered to a new consumer. As a consequence of this, there are cases where a message might be delivered more than once to multiple consumers. The application is responsible for converting the **at-least-once delivery** semantics to **exactly-once processing** semantics. As we have seen already, a typical way to achieve this is by associating every operation with a unique identifier that is used to deduplicate operations originating from the same message. Figure 8.4 shows an example of how this would work in practice[12].

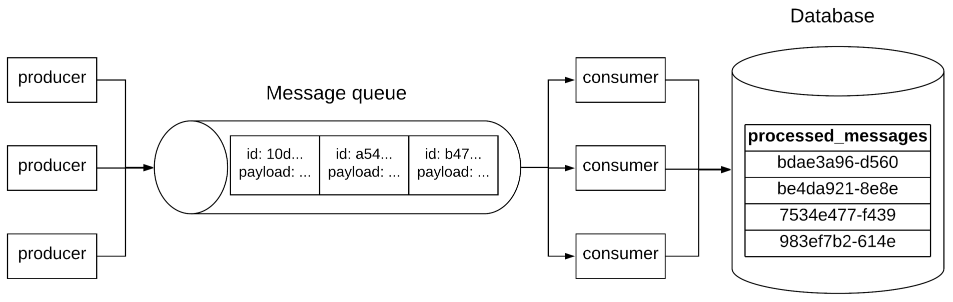


Figure 8.4: Example of exactly-once processing through deduplication

An event log provides a slightly different abstraction than a message queue. Messages are still inserted by producers at the tail of the log and stored in an ordered fashion. However, the consumers are free to select the point of the log they want to consume messages from, which is not necessarily the

[11]For example, Amazon SQS achieves that using a per-message visibility timeout, while ActiveMQ and RabbitMQ rely on a connection to timeout in order to redeliver all the unacknowledged messages of this connection.

[12]An important thing to note here is that the side-effect from the operation and the storage of the unique identifier must be done atomically to guarantee exactly-once processing. A simple way to do this is store both in the same datastore using a transaction.

head. Messages are typically associated with an index, which can be used by consumers to declare where they want to consume from. Another difference with message queues is that the log is typically immutable, so messages are not removed after they are processed. Instead, a garbage collection is run periodically that removes old messages from the head of the log. This means that consumers are responsible for keeping track of an offset indicating the part of the log they have already consumed in order to avoid processing the same message twice, thus achieving **exactly-once processing** semantics. This is done in a similar way as described previously with this offset playing the role of the unique identifier for each message. Some examples of event logs are Apache Kafka[13], Amazon Kinesis[14] and Azure Event Hubs[15].

Message queues and event logs can enable two slightly different forms of communication, known as **point-to-point** and **publish-subscribe**. The point-to-point model is used when we need to connect only 2 applications[16]. The publish-subscribe model is used when more than one applications might be interested in the messages sent by a single application. For example, the fact that a customer made an order might be needed by an application sending recommendations for similar products, an application that sends the associated invoices and an application that calculates loyalty points for the customer. Using a publish-subscribe model, the application handling the order is capable of sending a message about this order to all the applications that need to know about it, sometimes without even knowing which these applications are.

These two models of communication are implemented slightly different depending on whether an event log or a message queue is used. This difference is mostly due to the fact that consumption of messages behaves differently in each system:

- Point-to-point communication is pretty straightforward when using a message queue. Both applications are connected to a single queue, where messages are produced and consumed. In the publish-subscribe model, one queue can be created and managed by the producer application and every consumer application can create its own queue. There also needs to be a background process that receives messages from the

[13]See: https://kafka.apache.org

[14]See: https://aws.amazon.com/kinesis

[15]See: https://azure.microsoft.com/services/event-hubs

[16]Note that point-to-point refers to the number of applications, not the actual servers. Every application on each side might consist of multiple servers that produce and consume messages concurrently.

producer's queue and forwards them to the consumers' queues[17].
- When using an event log, both models of communication can be implemented in the same way. The producer application writes messages to the log and multiple consumer applications can be consuming from the log concurrently the same messages maintaining independent offsets.

Figure 8.5 illustrates this difference.

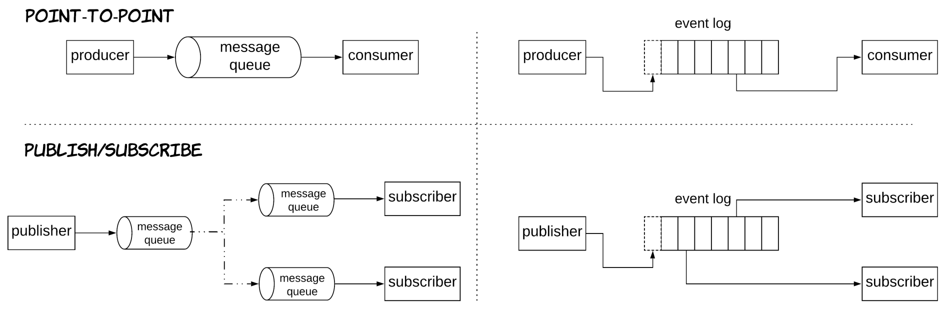


Figure 8.5: Implementations of point-to-point and publish-subscribe models

Coordination patterns

In many cases, a business function is performed by many different systems that cooperate with each other, so that each one of them performs some part of the overall function. For example, displaying a product page might require combining functionality from different systems, such as an advertising system, a recommendation system, a pricing system etc. The previous section examined the basic ways in which two different systems can communicate. This section will examine the two basic approaches that can be used to coordinate different systems in order to perform a common goal: **orchestration** and **choreography**.

In orchestration, there is a single, central system that is responsible for coordinating the execution of the various other systems, thus resembling a star topology. That central system is usually referred to as the *orchestrator* and it needs to have knowledge about all the other systems and their capabilities;

[17]Some systems might provide facilities that provide this functionality out of the box. For example, this is achieved in ActiveMQ via a *bridge* and in Amazon SQS via a separate AWS service, called Amazon Simple Notification Service (SNS).

these systems can be unaware of each other. In choreography, these systems coordinate with each other without the need of an external coordinator. They are essentially placed in a chain, where each system is aware of the previous and the next system in the topology. A request is successively passed through the chain from each system to the next one. Figure 8.6 contains a basic diagram of these two patterns.

Note that the communication link between two systems can be of any of the two forms described in the previous section. The systems can either communicate synchronously (i.e. using RPC calls) or asynchronously (i.e. via an intermediary message queue). Each option is subject to different trade-offs, some of which we will discuss in the next sections. These patterns will also behave slightly differently depending on whether the function performed has side-effects or not. For example, displaying a product page is most likely a function that does not have side-effects. This means that partial failures can be treated simply by retrying any failed requests, until they all succeed. Alternatively, if some requests are continuously failing, then they can be abandoned and the original request can be forced to fail. However, processing a customer order is most likely an operation with side-effects, which might need to be performed **atomically**. It's probably undesirable to charge a customer for an order that cannot be processed or to send an order to a customer that cannot pay. Each of the aforementioned patterns need to ensure this atomicity. One way to achieve this would be via a protocol like two-phase commit. An alternative would be to make use of the concept of saga transactions, described previously in the book. The first approach would fit better in the orchestrator pattern where the orchestrator plays the role of the coordinator of the two-phase commit protocol, while the second approach could be used in both patterns.

Data synchronisation

There are some cases, where the same data needs to be stored in multiple places and in potentially different forms[18]. Below are some typical examples:

- Data that reside in a persistent datastore also need to be cached in a separate in-memory store, so that read operations can be processed

[18]These are also referred to as *materialized views*. See: https://en.wikipedia.org/wiki/Materialized_view

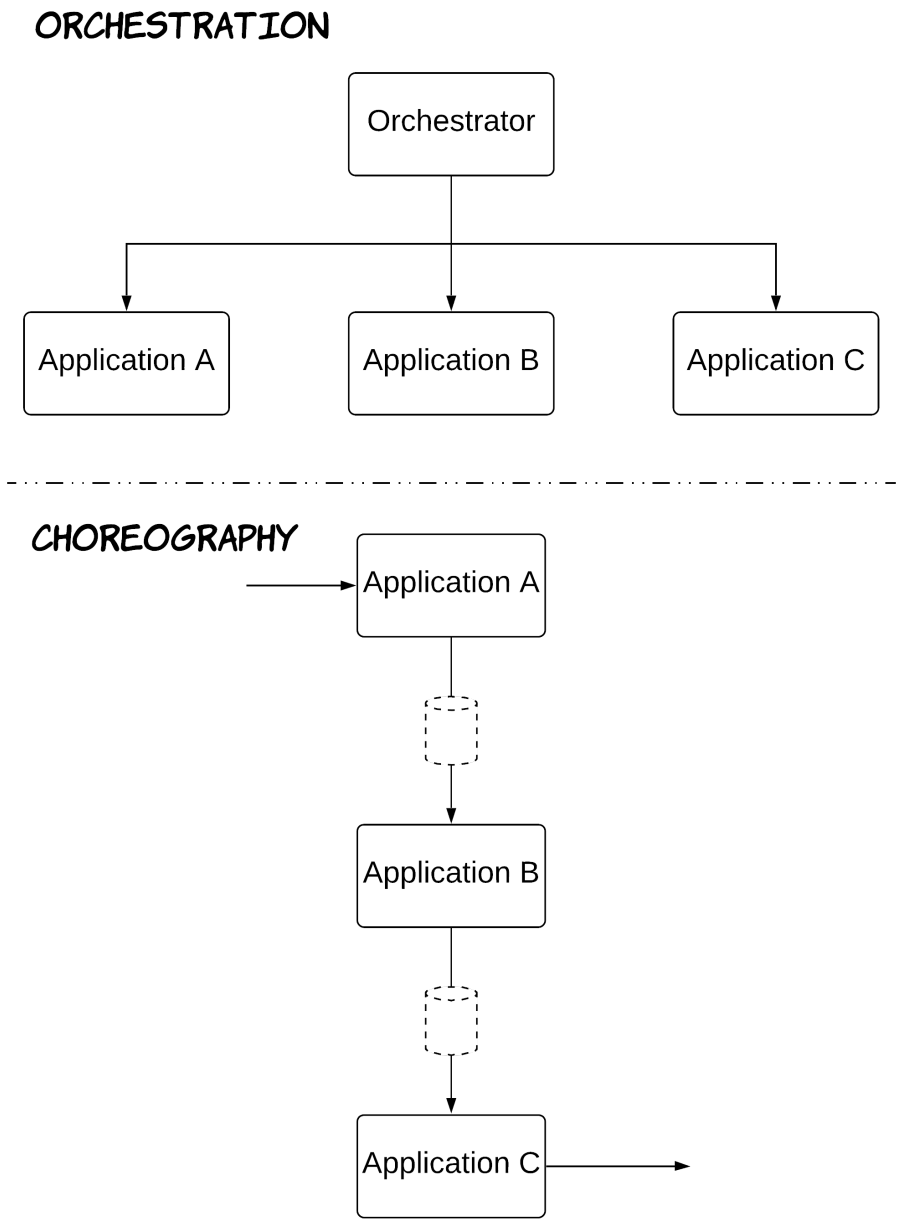


Figure 8.6: Orchestration and choreography

from the cache with lower latency. Writes operations need to update both the persistent datastore and the in-memory store.
- Data that are stored in a distributed key-value store need to also be stored in a separate datastore that provides efficient full-text search, such as ElasticSearch[19] or Solr[20]. Depending on the form of read operations, the appropriate datastore can be used for optimal performance.
- Data that are stored in a relational database need to also be stored in a graph database, so that graph queries can be performed in a more efficient way.

Given data reside in multiple places, there needs to be a mechanism that keeps them in sync. This section will examine some of the approaches available for this purpose and the associated trade-offs.

One approach is to perform writes to all the associated datastores from a single application that receives update operations. This approach is sometimes referred to as **dual writes**. Typically, writes to the associated data stores are performed synchronously, so that data have been updated in all the locations before responding to the client with a confirmation that the update operation was successful. One drawback of this approach is the way partial failures are handled and their impact on atomicity. If the application manages to update the first datastore successfully, but the request to update the second datastore fails, then atomicity is violated and the overall system is left in an inconsistent state. It's also unclear what the response to the client should be in this case, since data has been updated, but only in some places. However, even if we assume that there are not partial failures, there is another pitfall that has to do with how race conditions are handled between concurrent writers and their impact on isolation. Let's assume two writers submit an update operation for the same item. The application receives them and attempts to update both datastores, but the associated requests are reordered, as shown in Figure 8.7. This means that the first datastore contains data from the first request, while the second datastore contains data from the second request. This also leaves the overall system at an inconsistent state. An obvious solution to mitigate these issues is to introduce a distributed transaction protocol that provides the necessary atomicity & isolation, such as a combination of two-phase commit and two-phase locking. In order to be able to do this, the underlying datastores need to provide support for this. Even in that case though, this protocol will have some performance

[19]See: https://github.com/elastic/elasticsearch
[20]See: https://github.com/apache/lucene-solr

and availability implications, as explained in the previous chapters.

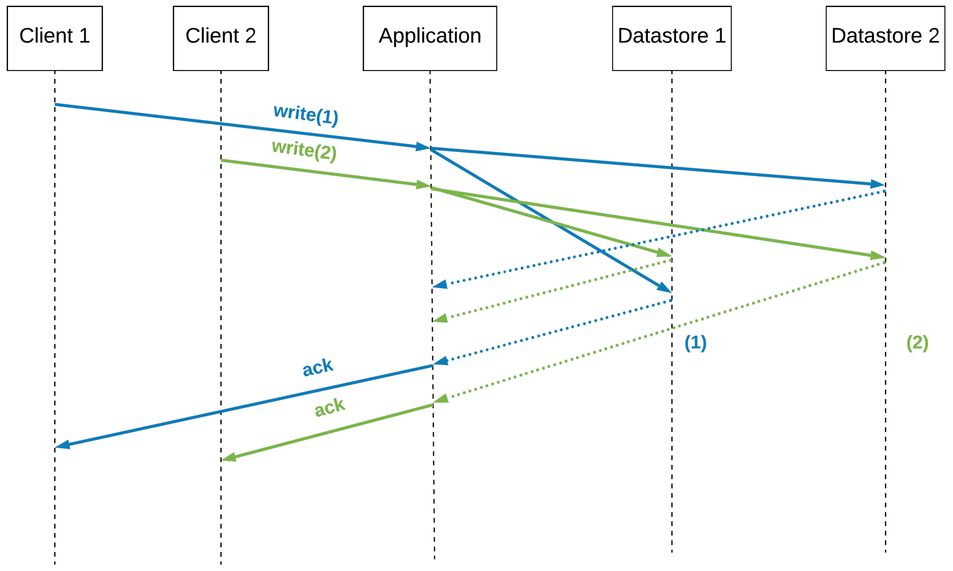


Figure 8.7: Isolation issues with dual writes

Another approach is writing any update operations to an append-only event log and having any interested applications consume events from this log and storing the associated data in their preferred datastore. This approach is called **event sourcing**. The current state of the system can be derived simply by consuming all the events from the beginning of the log. However, applications typically save periodical snapshots (also known as checkpoints) of the state to avoid having to reconsume the whole log in case of failures. In this case, an application that recovers from a failure only needs to replay the events of the log after the latest snapshot. This approach does not suffer from atomicity violations, which means there is no need for an atomic commit protocol. The reason is every application is consuming the log independently and they will eventually process all the events successfully, restarting from the last consumed event in case of a temporary failure. The isolation problem described in the first approach is also mitigated, since the applications will be consuming all the events in the same order. There is a small caveat: applications might be consuming the events from the log at different speeds, which means an event will not be reflected at the same instance on all the applications. This phenomenon can be handled at the application level. For example, if an item is not available in the cache, the application can query the other datastores. A different manifestation of this problem could be an item that has been indexed successfully has not been stored in the authoritative datastore yet, which could lead to a dangling pointer. This could also be mitigated at the application level, by identifying and discarding these items, instead of displaying broken links. If no such technique can be applied at the application level, a concurrency control protocol could be used, e.g. a locking protocol with the associated performance and availability costs.

Some kind of applications need to perform update operations that need an up-to-date view of the data. The simplest example is a conditional update operation[21], create a user if no user with the same username exists already. This is not easily achievable when using event sourcing, because the applications consume the log asynchronously, so they are only *eventually consistent*. There is another approach that solves this problem, known as **change data capture** (CDC). When using this approach, a datastore is selected as the authoritative source of data, where all update operations are performed. An event log is then created from this datastore that is consumed by all the remaining operations the same way as in event sourcing. This primary datastore needs to provide the necessary transactional semantics and a way to monitor changes in the underlying data in order to produce the event log. Relational databases are usually a good fit for this, since most of them provide strong transactional guarantees and they internally use a write-ahead-log (WAL) that imposes an order on the performed operations and can be used to feed an event log[22]. As a result, clients are able to perform updates that are conditional on the current state and the remaining applications can apply these operations independently at their own datastores.

Shared-nothing architectures

At this point, it must have become evident that sharing leads to coordination, which is one of the main factors that inhibit high availability, performance and scalability. For example, we have already explained how distributed databases can scale to larger datasets and in a more cost-efficient manner than centralised, single-node databases. At the same time, some form of sharing is sometimes necessary and even beneficial for the same characteristics. For instance, a system can increase its overall availability by reducing sharing

[21]This is also known as a compare-and-set (CAS) operation. See: https://en.wikipedia.org/wiki/Compare-and-swap

[22]An example of a tool that does this is Debezium. See: https://debezium.io

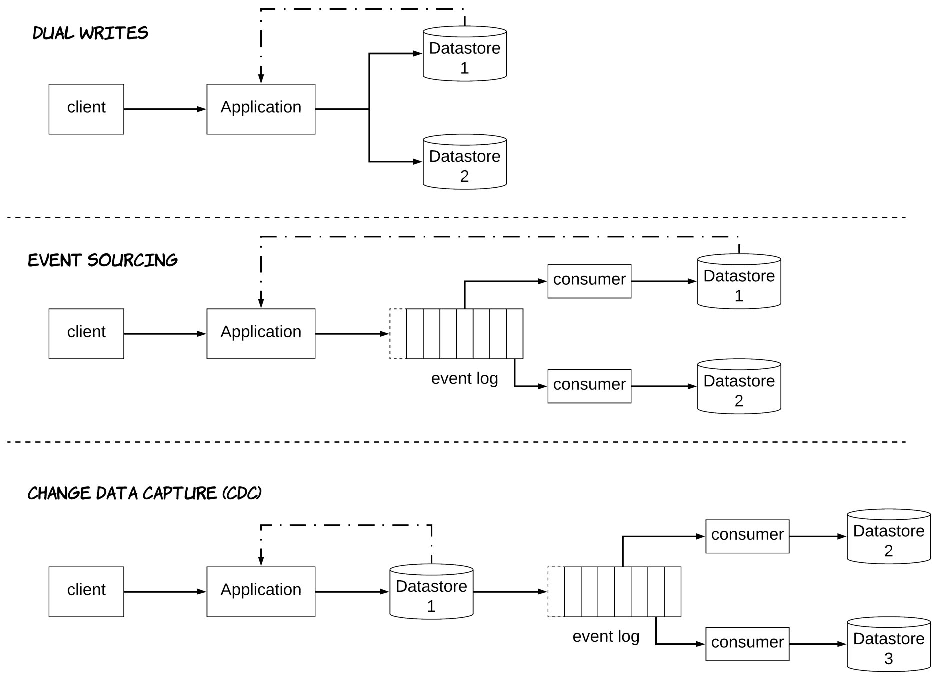


Figure 8.8: Data synchronisation techniques

through partitioning, since the various partitions can have independent failure modes. However, when looking at a single data item, availability can be increased by increasing sharing via replication.

A key takeaway from all this is that reducing sharing can be very beneficial, when applied properly. There are some system architectures that follow this principle to the extreme in order to reduce coordination and contention, so that every request can be processed independently by a single node or a single group of nodes in the system. These are usually called **shared-nothing** architectures. This section will explain how this principle can be used in practice to build such architectures and what are some of the trade-offs.

A basic technique to reduce sharing used widely is decomposing **stateful** and **stateless** parts of a system. The main benefit from this is that stateless parts of a system tend to be fully symmetric and homogeneous, which means that every instance of a stateless application is indistinguishable from the rest. Separating them makes scaling a lot easier. Since all the instances of an application are identical, one can balance the load across all of them in an easy way, since all of them should be capable of processing any incoming request[23]. The system can be scaled out in order to handle more load by simply introducing more instances of the applications behind the load balancer. The instances could also send heartbeats to the load balancer, so that the load balancer is capable of identifying the ones that have potentially failed and stop sending requests to them. The same could also be achieved by the instances exposing an API, where requests can be sent periodically by the load balancer to identify healthy instances. Of course, in order to achieve high availability and be able to scale incrementally, the load balancer also needs to be composed of multiple, redundant nodes. There are different ways to achieve this in practice, but a typical implementation uses a single domain name (DNS) for the application that resolves to multiple IPs belonging to the various servers of the load balancer. The clients, such as web browsers or other applications, can rotate between these IPs. The DNS entry needs to specify a relatively small time-to-live (TTL), so that clients can identify new servers in the load balancer fleet relatively quickly.

As seen already throughout the book, stateful systems are a bit harder to manage, since the various nodes of the system are not identical. Each node contains different pieces of data, so the appropriate routing must be

[23]Note that in practice the load balancer might need to have some state that reflects the load of the various instances. But, this detail is omitted here on purpose for the sake of simplicity.

performed in order to direct requests to the proper part of the system. As implied before, the presence of this state also creates a tension that prevents us from completely eliminating sharing if there is a need for high availability. A combination of partitioning and replication is typically used to strike a balance. Data are partitioned to reduce sharing and create some independence and fault isolation, but every partition is replicated across multiple nodes to make every partition fault tolerant. We have already examined several systems that follow this pattern, such as Cassandra and Kafka. Sharing is thus not a binary property of a system, but rather a spectrum. On the one end of the spectrum, there might be systems that have as little sharing as possible, i.e. not allowing any transactions across partitions in order to reduce the coordination needed. Some systems might fall in the middle of the spectrum, where transactions are supported, but performed in a way that introduces coordination only across the involved partitions instead of the whole system. Lastly, systems that store all the data in a single node fall somewhere on the other end of the spectrum.

Figure 8.9 illustrates an example of such a shared-nothing architecture. As explained already, such an architecture provides a lot of benefits from a performance and fault-tolerance perspective. All layers of the applications can be incrementally scaled out or in depending on the load. Admittedly, this is easier and quicker to achieve in the stateless components, since it requires less data transfer. The system is resilient to single-node and multi-node failures. More specifically, these two different forms of failure impact the stateless parts of the system in a similar way, the size of the impact is just different. For example, the remaining nodes might need to handle bigger load or more servers might need to be provisioned. For the stateful parts of the architecture, these two different forms of failure have slightly different behaviours. Single-node failures are a lot easier to handle, since each partition can use a consensus-based technique for replication which can remain fully functional as long as a majority of nodes is healthy. However, multi-node failures can affect a majority, thus making a partition unavailable. Even in this case, the good thing is only this partition will be unavailable and the rest of system's data will still be available.

This type of architecture tends to be a good fit for problems that are amenable to fine-grained partitioning. Some examples of problems in this space are managing user sessions or managing the products of a catalog. In both cases, data can easily be partitioned in a way, where data access operations will need to access a single data item. For example, sessions can be assigned a unique identifier and they can be partitioned by this attribute and products

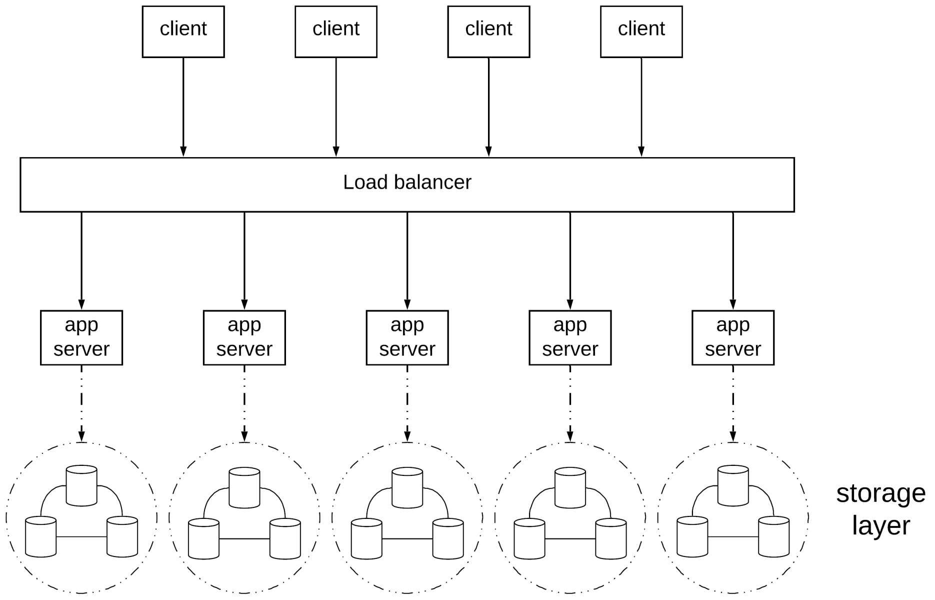


Figure 8.9: Example of a typical shared-nothing architecture

can be partitioned according to a similar product identifier. If sessions and products are mostly retrieved and updated by their identifier, this can be done quite efficiently by querying only the nodes that have the associated data. There is also space for some form of limited concurrency control in the scope of a partition. For cases where single-item access is the norm, a common technique is using **optimistic concurrency control** in order to reduce overhead and contention. This can be achieved by including a `version` attribute on every data item. Every writer performs a read before a write in order to find the current version and then includes the current version in the update as a condition to be satisfied in order for it to be completed. Of course, this requires the corresponding datastore to provide support for conditional updates. If a concurrent writer has updated the same item in the meanwhile, the first writer will have to abort and restart by performing a new read to determine whether its initial write should still apply and if so, retry it.

Of course, all of this does not mean this architecture does not have drawbacks. The main one is reduced flexibility. If the application needs access to new data access patterns in an efficient way, it might be hard to provide it given the system's data have been partitioned in a specific way. For example, attempting to query by a secondary attribute that is not the partitioning key might require to access all the nodes of the system. This reduced flexibility might also manifest in lack of strong transactional semantics. Applications that need to perform reads of multiple items with strong isolation or write multiple items in a single atomic transaction might not be able to do this under this form of architecture or it might only be possible at the cost of excessive additional overhead.

Distributed locking

As explained already in the introductory chapters of the book, concurrency is one of factors that contribute significantly to the complexity of distributed systems. A mechanism is needed to ensure that all the various components of a distributed system that are running concurrently do so in a way that is safe and does not bring the overall system to an inconsistent state. An example we have already seen is leader election, where the system needs to ensure only one node in the system is capable of performing the leader duties at any point in time. Amongst the available techniques, locking is the simplest solution and one that is used commonly. However, locking techniques are subject to different failure modes when applied in a distributed system. This section will cover some common pitfalls and how to address them to use locking safely in a distributed system.

The main property derived from the use of locks is **mutual exclusion**: multiple concurrent actors can be sure that only one of them will be performing a critical operation at a time. Typically, all of the actors have to follow the same sequence of operations, which is first acquiring the lock, performing that critical operation and then releasing the lock, so that other workers can proceed. This is usually simple to implement in cases where all actors are running inside the same application sharing a single memory address space and the same lifecycle. However, doing the same in a distributed system is much more complicated, mostly due to the potential of partial failures.

The main complication in a distributed system is that the various nodes of the system can fail independently. As a result, a node that is currently holding a lock might fail before being able to release the lock. This would bring the system to a halt until that lock is released via some other means (i.e. via an operator), thus reducing availability significantly. A timeout mechanism can be used to cope with this issue. A **lease**[84] is essentially a lock with an expiry timeout after which the lock is automatically released by the system that is responsible for managing the locks. By using leases instead of locks, the system can automatically recover from failures of nodes that have acquired locks by releasing these locks and giving the opportunity to other nodes to acquire them in order to make progress. However, this introduces new safety risks. There are now two different nodes in the system that can have different views about the state of the system, specifically which nodes holds a lock. This is not only due to the fact that these nodes have different clocks so the time of expiry can differ between them, but also because a failure detector cannot be perfect, as explained earlier in the book. The fact that part of the system considers a node to be failed does not mean this node is necessarily failed. It could be a network partition that prevents some messages from being exchanged between some nodes or that node might just be busy with processing something unrelated. As a result, that node might think it still holds the lock even though the lock has expired and it has been automatically released by the system.

Figure 8.10 shows an example of this problem, where nodes A and B are trying to acquire a lease in order to perform some operations in a separate system. Node A manages to successfully acquire a lease first. However, there is a significant delay between acquiring the lease and performing the associated operation. This could be due to various reasons, such as a long garbage collection pause, scheduling delays or simply network delays. In the meanwhile, the lease has expired, it has been released by the system and acquired by node B, which has also managed to perform the operation that's protected by the lock. After a while, the operation from node A also reaches the system and it's executed even though the lease is not held anymore by that node violating the basic invariant that was supposed to be protected by the lease mechanism. Note that simply performing another check the lease is still held before initiating the operation in node A would not solve the problem, since the same delays can occur between this check and the initiation of the operation or even the delivery of the operation to the system.

There is one simple technique that solves this problem and it's called **fencing**. The main idea behind fencing is that the system can block some nodes from performing some operations they are trying to perform, when these nodes are malfunctioning. In our case, nodes are malfunctioning in the sense that they think they hold a lease, while they don't. The locking subsystem can associate every lease with a monotonically increasing number. This number can then be used by all the other systems in order to keep track of the node

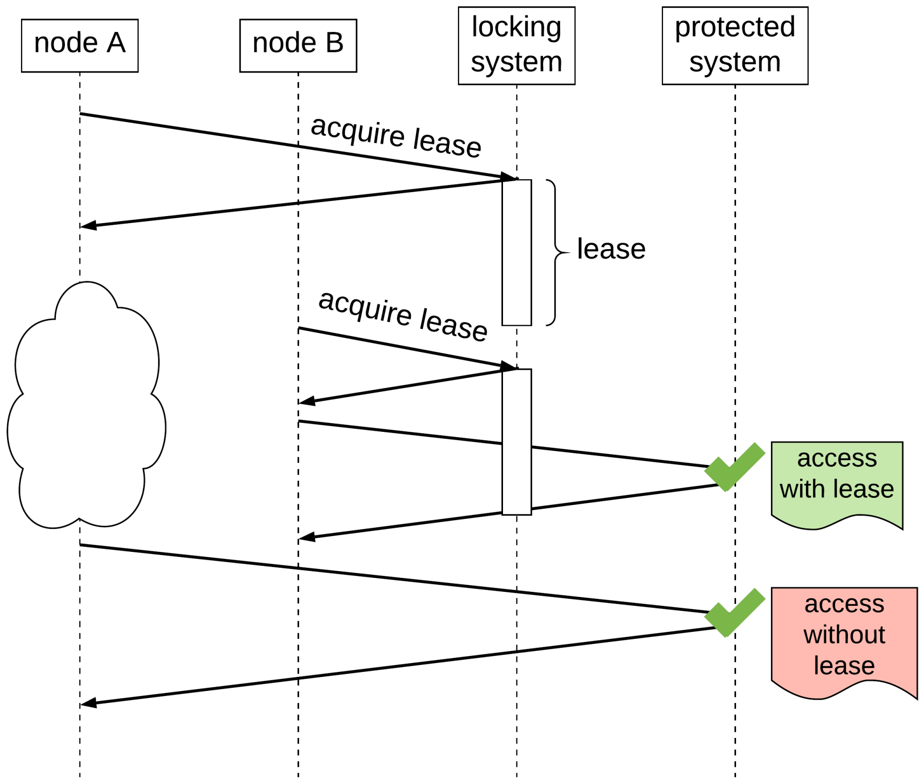


Figure 8.10: Issues with distributed locking & leases

that has performed an operation with the most recent lease. If a node with an older lease attempts to perform an operation, the system is able to detect that and reject the operation, while also notifying the node that it's not the leader anymore. Figure 8.11 shows how that would work in practice. This essentially means that in a distributed system, lock management cannot be performed by a single part of the system, but it has to be done collectively by all the parts of the system that are protected by this lock. For this to be possible, the various components of the system need to provide some basic capabilities. The locking subsystem needs to provide a monotonically increasing identifier for every lock acquisition. Some examples of systems that provide this is Zookeeper via the `zxid` or the znode version number and Hazelcast as part of the fenced token provided via the `FencedLock` API. Any external systems protected by the locks needs to provide conditional updates with linearizability guarantees.

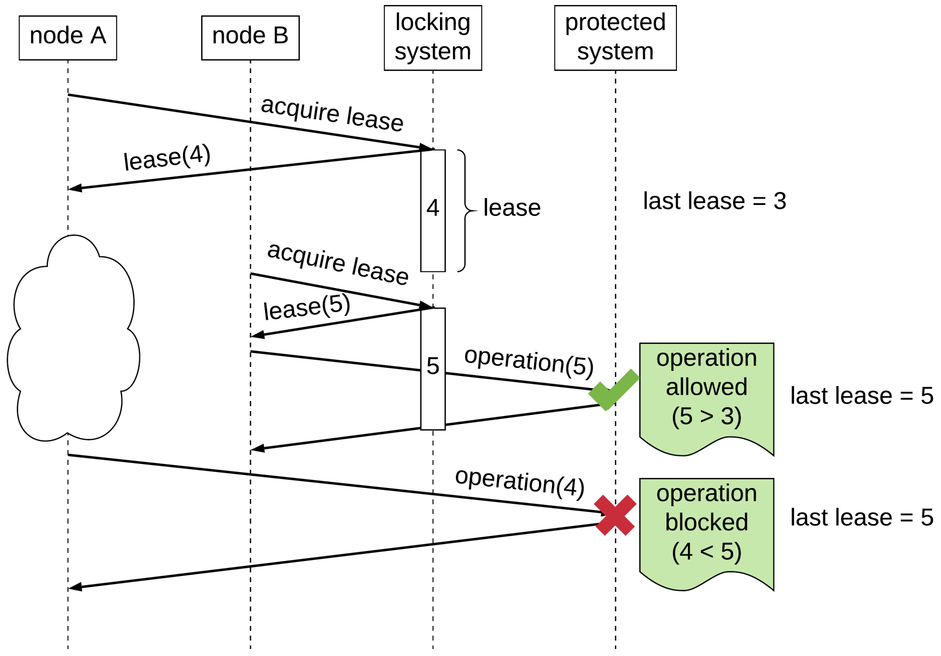


Figure 8.11: Mutual exclusion using fencing

Compatibility patterns

As explained early on in the book, a defining characteristic of distributed systems is they are composed of multiple nodes. In general, it is useful to allow the various nodes of such a system to operate independently for various reasons. A very typical requirement for some applications in real life is to be able to deploy new versions of the software with zero downtime. The simplest way to achieve that is to perform rolling deployments, instead of deploying in lockstep the software to all the servers at the same time. In some cases, this is not just a nice-to-have, but an inherent characteristic of the system. An example of this is mobile applications (e.g. Android applications), where user consent is required to perform an upgrade, which implies that users are deploying the new version of the software at their own pace. As a result, the various nodes of a distributed systems can be running different versions of the software at any time. This section will examine the implications of this and some techniques that can help manage this complication.

The phenomenon described previously manifests in many different ways. One of the most common ones is when two different applications need to communicate with each other, while each one of them evolves independently by deploying new versions of its software. For example, one of the applications might want to expose more data at some point. If this is not done in a careful way, the other application might not be able to understand the new data making the whole interaction between the applications fail. Two very useful properties related to this are **backward compatibility** and **forward compatibility**:

- Backward compatibility is a property of a system that provides interoperability with an earlier version of itself or other systems.
- Forward compatibility is a property of system that provides interoperability with a later version of itself or other systems.

These two properties essentially reflect a single characteristic viewed from different perspectives, those of the sender and the recipient of data. Let's consider for a moment a contrived example of two systems `S` and `R`, where the former sends some data to the latter. We can say that a change on system `S` is backward compatible, if older versions of `R` will be able to communicate successfully with a new version of `S`. We can also say that the system `R` is designed in a forward compatible way, so that it will be able to understand new versions of `S`. Let's look at some examples. Let's assume system `S` needs to change the data type of a specific attribute. Changing the data type of an existing attribute is not a backward compatible change in general, since system `R` would be expecting a different type for this attribute and it would fail to understand the data. However, this change could be decomposed in the following sub-changes that preserve compatibility between the two systems. The system `R` can deploy a new version of the software that is is capable of reading that data either from the new attribute with the new data type or the old attribute with the old data type. The system `S` can then deploy a new version of the software that stops populating the old attribute and starts populating the new attribute[24]. The previous example probably demonstrated that seemingly trivial changes to software are a lot more complicated when they need to be performed in a distributed system in a safe way. As a consequence, maintaining backward compatibility imposes a tradeoff between agility and safety.

It's usually beneficial to version the API of an application, since that makes it easier to compare versions of different nodes and applications of the system and determine which versions are compatible with each other or not. **Semantic versioning** is a very useful convention, where each version number consists of 3 digits `x.y.z`. The last one (`z`) is called the patch version and it's incremented when a backward compatible bug fix is made to the software. The second one (`y`) is called the minor version and it's incremented when new functionality is added in a backward compatible way. The first one (`x`) is called the major version and it's incremented when a backward incompatible change is made. As a result, the clients of the software can easily quickly understand the compatibility characteristics of new software and the associated implications. When providing software as a binary artifact, the version is usually embedded in the artifact. The consumers of the artifact then need to take the necessary actions, if it includes a backward incompatible change, e.g. adjusting their application's code. However, when applied to live applications, semantic versioning needs to be implemented slightly differently. The major version needs to be embedded in the address of the application's endpoint, while the major and patch versions can be included in the application's responses[25]. This is needed so that clients can be automatically upgraded to newer versions of the software if desired, but only if these are backward compatible.

[24]Note that whether a change is backward compatible or not can differ depending on the serialization protocol that is used. The following article explains this nicely: https://martin.kleppmann.com/2012/12/05/schema-evolution-in-avro-protocol-buffers-thrift.html

[25]For an example of this, see: https://developers.facebook.com/docs/apps/versions

Another technique for maintaining backward compatibility through the use of explicitly versioned software is **protocol negotiation**. Let's assume a scenario as mentioned previously, where the client of an application is a mobile application. Every version of the application needs to be backward compatible with all the versions of the client application running on user phones currently. This means that the staged approach described previously cannot be used when making backward incompatible changes, since end users cannot be forced to upgrade to a newer version. Instead, the application can identify the version of the client and adjust its behaviour accordingly. For example, consider the case of a feature introduced on version `4.1.2` that is backward incompatible with versions < `4.x.x`. If the application receives a request from a `3.0.1` client, it can disable that feature in order to maintain compatibility. If it receives a request from a `4.0.3` client, it can enable the feature.

In some cases, an application might not be aware of the applications that will be consuming its data. An example of this is the publish-subscribe model, where the publisher does not necessarily need to know all the subscribers. It's still very important to ensure consumers will be able to deserialise and process any produced data successfully as its format changes. One pattern used here is defining a schema for the data, which is used by both the producers and consumers. This schema can be embedded in the message itself. Otherwise, to avoid duplication of the schema data, a reference to the schema can be put inside the message and the schema can be stored in a separate store. For example, this is a pattern commonly used in Kafka via the Schema Registry[26]. However, it's important to remember that even in this case, producers and consumers are evolving independently, so consumers are not necessarily using the latest version of the schema used by the producer. So, producers need to preserve compatibility either by ensuring consumers can read data of the new schema using an older schema or by ensuring all consumers have started using the new schema before starting to produce messages with it. Note that similar considerations need to be made for the compatibility of the new schema with old data. For example, if consumers are not able to read old data with the new schema, the producers might have to make sure all the messages with the previous schema have been consumed by everyone first. Interestingly, the Schema Registry defines different categories of compatibility along these dimensions, which determine what changes are allowed in each category and what is the upgrade process, e.g. if producers or consumers need to upgrade first. It can also check two different versions of

[26]See: https://docs.confluent.io/current/schema-registry

a schema and confirm that they are compatible under one of these categories to prevent errors later on[27].

Note that it's not only changes in data that can break backward compatibility. Slight changes in behaviour or semantics of an API can also have serious consequences in a distributed system. For example, let's consider a failure detector that makes use of heartbeats to identify failed nodes. Every node sends a heartbeat every 1 second and the failure detector considers a node failed if it hasn't received a single heartbeat in the last 3 seconds. This causes a lot of network traffic that affects the performance of the application, so we decide to increase the interval of a heartbeat from 1 to 5 seconds and the threshold of the failure detector from 3 to 15 seconds. Note that if we start performing a rolling deployment of this change, all the servers with the old version of the software will start thinking all the servers with the new version have failed. This due to the fact that their failure detectors will still have the old deadline of 3 seconds, while the new servers will send a heartbeat every 5 seconds. One way to make this change backward compatible would be to perform an initial change that increases the failure detector threshold from 3 to 15 seconds and the follow this with a subsequent change that increases the heartbeat interval to 5 seconds, only after the first change has been deployed to all the nodes. This technique of splitting a change in two parts to make it backward compatible is commonly used and it's also known as **two-phase deployment**.

Dealing with failure

Failure is the norm in a distributed system, so building a system that is able to cope with failures is crucial. This section will cover some principles on how to deal with failures and present some basic patterns for building systems that are resilient to failures. As shown throughout this section, dealing with a failure consists of three main parts: **identifying** the failure, **recovering** from the failure and in some cases **containing** a failure to reduce its impact.

Hardware failures can be the most damaging ones, since they can lead to data loss or corruption. On top of that, the probability of a hardware failure is significantly higher in a distributed system due to the bigger number of hardware components involved. Silent hardware failures are the ones with the

[27]See: https://docs.confluent.io/current/schema-registry/avro.html#schema-evolution-and-compatibility

biggest impact, since they can potentially affect the behaviour of a system without anyone noticing. An example of a silent failure would be a node in the network corrupting some part of a message, so that the recipient receives data that is different to what the sender originally sent without being able to detect that. Similarly, data written to a disk can be corrupted during the write operation or even a long time after that was completed. Below are some techniques that are commonly used to handle these kind of failures:

- One way to detect these failures when sending a message to another node is to introduce some **redundancy** in the message using a **checksum** derived from the actual payload. If the message is corrupted, the checksum will not be valid. As a result, the recipient can ask the sender to send the message again.
- When writing data to disk, this technique might not be useful, since the corruption will be detected a long time after a write operation has been performed by the client, which means it might not be feasible to rewrite the data. Instead, the application can make sure that data is written to multiple disks, so that corrupted data can be discarded later on and the right data can be read from another disk with a valid checksum.
- Another technique used in cases where retransmitting or storing the data multiple times is impossible or costly is **error correcting codes** (ECC). These are similar to checksums and are stored alongside the actual payload, but they have the additional property that they can also be used to correct corruption errors calculating the original payload again. The downside is they are larger than checksums, thus having a higher overhead in terms of data stored or transmitted across the network.

A distributed system consists of many different parts and these kind of failures can happen on any of them. This raises the question of where and how to apply these techniques. There is a design principle, known as the **end-to-end argument**, which suggests that some functions such as the fault tolerance techniques described above can be implemented completely and correctly only wih the knowledge and help of the application standing at the end points of the communication system. A canonical example to illustrate this point is the "careful file transfer" application, where a file needs to be moved from computer A's storage to computer B's storage without damage. As shown in Figure 8.12, hardware failures can happen in many places during this process, such as the disks of computers, the software of the file system, the hardware processors, their local memory or the communication system.

Even if the various subsystems embed error recovery functionality, this can only cover lower levels of the system and it cannot protect from errors happening at a higher level of the system. For example, error detection and recovery implemented at the disk level or in the operating system won't help, if the application has a defect that leads to writing the wrong data in the first place. This implies that complete correctness can only be achieved by implementing this function at the application level. Note that this function can be implemented redundantly at lower levels too, but this is done mostly as a performance optimisation. It's also important to note that this redundant implementation at lower levels is not always beneficial, but it depends on the use case. There is existing literature that covers extensively this trade-off, so we'll refer the reader to it instead of repeating the same analysis here[85][86].

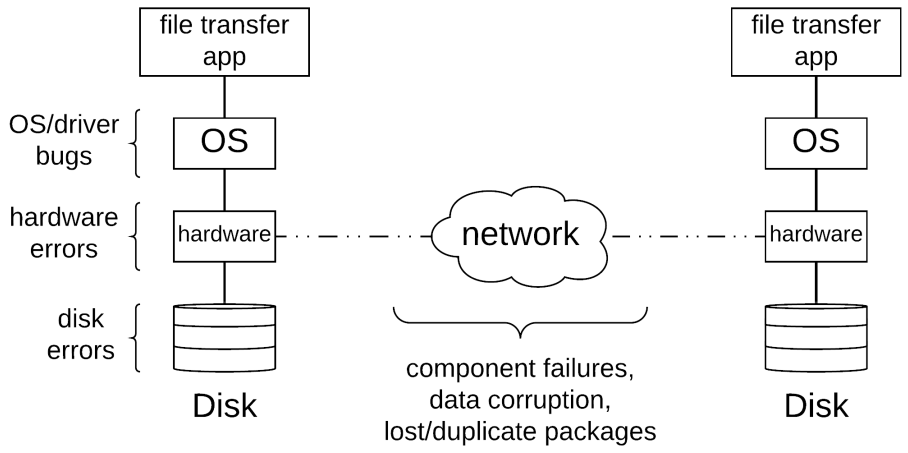


Figure 8.12: Careful file transfer and possible failures

It's interesting to observe that this principle manifests in many different ways when dealing with a distributed system. The most relevant problem we have encountered repeatedly throughout this book is providing **exactly-once** guarantees. Let's consider an extremely simplified version of the problem, where applications A wants to trigger an operation on application B exactly once and each application consists of a single server. Note that the communication subsystem, specifically TCP, can provide reliable delivery of data via retries, acknowledgements and deduplication, which are the techniques already described in this book. However, this is still not sufficient for providing exactly-once guarantees at the application level. Let's look at some of the things that can go wrong:

- The TCP layer on the side of application B might receive a packet and acknowledge it back to the sender side, while buffering it locally to be delivered to the application. However, the application crashes before it manages to receive this packet from the TCP layer and process it. In this case, the application will think the packet has been successfully processed, while it wasn't.
- The TCP layer on the side of the application B might receive a packet and deliver it successfully to the application, which processes it successfully. However, a failure happens at this point and the applications on both sides are forced to establish a new TCP connection. Application had not received an application acknowledgement for the last message, so it attempts to resend it on the new connection. TCP provides reliable transfer only in the scope of a single connection, so it will not be able to detect this packet has been received and processed in a previous connection. As a result, a packet will be processed by the application more than once.

The main takeaway is that any functionality needed for exactly-once semantics (e.g. retries, acknowledgements and deduplication) needs to be implemented at the application level in order to be correct and safe against all kind of failures[28]. Another problem where the end-to-end principle manifests in a slightly different shade is the problem of mutual exclusion in a distributed system. The fencing technique presented previously essentially extends the function of mutual exclusion to all the involved ends of the application. The goal of this section is not to go through all the problems, where the end-to-end argument is applicable. Instead, the goal is to raise awareness and make the reader appreciate its value on system design, so that it's taken into account if and when need be.

The main technique to recover from failures is using **retries**. In the case of a stateless system, the application of retries is pretty simple, since all the nodes of the application are identical from the perspective of the client so it could retry a request on any node. In some cases, that can be done in a fully transparent way to the client. For example, the application can be fronted by a load balancer that receives all the requests under a single domain and it's responsible for forwarding the requests to the various nodes

[28]It's also worth reminding here that the side-effects from processing a request and storing the associated deduplication ID need to be done in an atomic way to avoid partial failures violating the exactly-once guarantees.

of the application. In this way, the client would only have to retry the request to the same endpoint and the load balancer would take care of balancing the requests across all the available nodes. In the case of stateful systems, this gets slightly more complicated, since nodes are not identical and retries need to be directed to the right one. For example, when using a system with master-slave replication, a failure of the master node must be followed by a failover to a slave node that is now the new master and new requests should be going there. There are different mechanisms to achieve this depending on the technology used. The same applies to consensus-based replication, where new leader election might need to happen and write operations need to be directed to the current leader.

Most of the techniques described so far are used to identify failure and recover from it. It's also useful to be able to contain the impact of a failure, so we will now discuss some techniques for this purpose. This can be done via technical means, such as **fault isolation**. One common way to achieve this is to deploy an application redundantly in multiple facilities that are physically isolated and have independent failure modes. So, when there is an incident that affects one of these facilities, the other facilities are not impacted and continue functioning as normal. Note that this introduces a trade-off between availability and latency, since physical isolation comes with increased network distance and latency. Most of the cloud providers provide multiple datacenters that are physically isolated and all located close to each other in a single region to strike a good balance in this trade-off. These are commonly known as *availability zones*. There are also cases where this can be achieved by a technique called **graceful degradation**, where an application reduces the quality of its service in order to avoid failing completely. For instance, let's think about a service that provides the capabilities of a search engine by calling to downstream services. Let's also assume one of these services that provides the advertisements to be shown for each query is having some issues. The top-level service can just render the results of a search term without any advertisements, instead of returning an error message or a completely empty response.

Techniques to contain failure can be broadly categorised in two main groups: those performed at the client side and those performed at the server side. A very useful concept in the field of distributed systems is backpressure. **Backpressure** is essentially a resistance to the desired flow of data through a system. This resistance can manifest in different ways, such as increased latency of requests or failed requests. Backpressure can also be implicit or explicit. For example, implicit backpressure arises in a system that is overloaded by a traffic surge and becomes extremely slow. On the other hand, a system that rejects some requests during a traffic surge in order to maintain a good quality of service is essentially exerting explicit backpressure to its clients. Most of the techniques to contain failure reflect how applications exert backpressure and how their clients handle backpressure.

Let's look first at how applications can exert backpressure. It is useful for a system to know its limits and exert backpressure when they are reached, instead of relying on implicit backpressure. Otherwise, there can be many failure modes that are unexpected and harder to deal with when they happen. The main technique to exert backpressure is **load shedding**, where an application is aware of the maximum load it can handle and rejects any requests that cross this threshold in order to keep operating at the desired levels. A more special form of load shedding is **selective client throttling**, where an application assigns different quotas to each of its clients. This technique can also be used to prioritise traffic from some clients that are more important. Let's consider a service that is responsible for serving the prices of products, which is used both by systems that are responsible to display product pages and by systems that are responsible to receive purchases and charge the customer. In case of a failure, that service could throttle the former type of systems more than the latter, since purchases are considered to be more important for a business and they also tend to constitute a smaller percentage of the overall traffic. In the case of asynchronous systems that make use of message queues, load shedding can be performed by imposing an upper bound on the size of the queue. There is a common misconception that message queues can help absorb any form of backpressure without any consequences. However, this comes at the cost of an increased backlog of messages, which can lead to increased processing latency or even failure of the messaging system in extreme cases.

Clients of an application should also be able to react properly to backpressure emitted by an application. The most typical way to react to failures in a distributed systems is **retries**. This is done under the assumption that a failure is temporary, so retrying a request is expected to have a better outcome. However, retries can have adverse effects, such as overloading a service. There are some techniques to make sure retries are used properly and negative side-effects are avoided as much as possible. First of all, it is useful to think about the whole architecture of the systems and the various applications involved to determine where retries will be performed. Performing retries at multiple levels can lead to significant amplification of the traffic coming from customers, which can overload services and cause

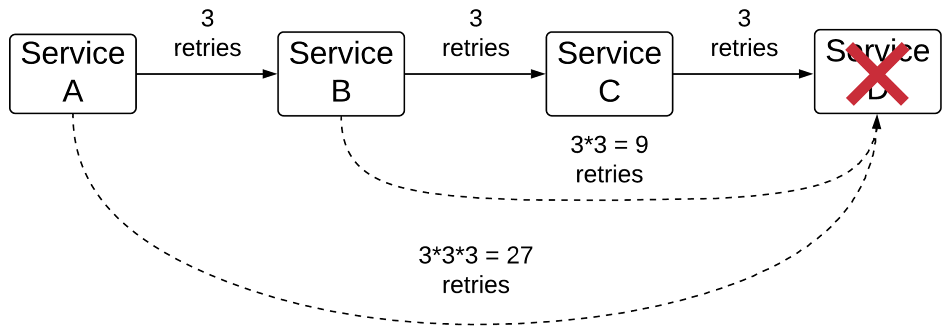


Figure 8.13: Load amplification due to retries

issues. As an example, let's assume we have 4 services A, B, C and D that call each other in order, as shown in Figure 8.13. If each service performs 3 retries for every failed request, then a temporary issue at service D will cause every request to be retried 27 times, thus creating a lot of additional load to service D during a period it's already experiencing issues. A conventional approach to mitigate this issue is to retry failed requests at the highest level possible, which usually contains additional context around the business function of the request and whether it's actually worth retrying or not. Another technique is using **exponential backoff** when retrying a request, so that the system waits a bit more every time before performing the next retry. This gives the downstream system a better opportunity to recover from any temporary issues. Ideally, exponential backoff is also combined with some **jitter**. This is so retries from various servers of a service are distributed evenly and they do not produce sudden spikes of traffic that can also cause overload issues. Clients of an application can also perform some form of load shedding to help downstream applications to recover with the use of **circuit breaker**. A circuit breaker essentially monitors the percentage of failed requests. When a specific threshold is crossed, this is interpreted as a permanent failure of the downstream application. As a result, the circuit breaker rejects all the requests locally without sending them to the downstream application. The circuit breaker allows just a few requests to be sent periodically and if a good percentage of them is successful, it starts sending load again. This technique is beneficial in two ways. First of all, it gives the downstream service a chance to recover from overload situations or other kinds of permanent failures. On top of that, it improves the customer experience by reducing request latency in cases where the response from a downstream is not absolutely necessary[29]. Another useful thing clients can do to help downstream applications is embed **timeout hints** in their requests. These hints inform downstream applications about the time after which a response to a request is not useful anymore. In this way, downstream applications can simply discard requests that had been waiting for a long time in message queues or in memory buffers due to resource exhaustion, thus speeding up processing of accumulated backlogs.

Distributed tracing

Tracing refers to the special use of logging to record information about a program's execution, which can be used for troubleshooting or diagnostic purposes. In its simplest form, this can be achieved by associating every request to the program with a unique identifier and recording logs for the most important operations of the program alongside the request identifier. In this way, when trying to diagnose a specific customer issue, the logs can easily be filtered down to only include a chronologically ordered list of operation logs pertaining to the associated request identifier. These logs can provide a summary of the various operations the program executed and the steps it went through.

In a distributed system, every client request is typically served through the use of multiple, different applications. As a result, one needs to collate traces from multiple programs in order to fully understand how a request was served and where something might have gone wrong. This is not as easy as it sounds, because every application might be using its own request identifiers and the applications are most likely processing multiple requests concurrently. This makes it harder to determine which requests correspond to a specific client request. This problem can be solved through the use of **correlation identifiers**. A correlation identifier is a unique identifier that corresponds to a top-level client request. This identifier might be automatically generated or it might be provided by some external system or manual process. It is then propagated through the various applications that are involved in serving this request. These applications can then include that correlation identifier in their logs along with their own request identifiers. In this way, it is easier to identify all the operations across all applications corresponding

[29]An example of this is the situation described previously, when explaining the concept of graceful degradation.

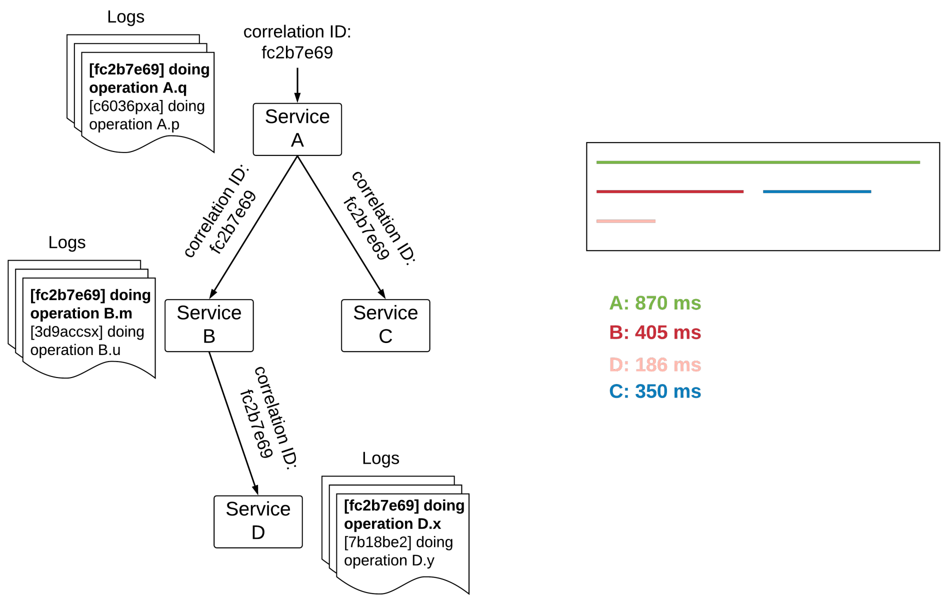


Figure 8.14: Distributed tracing via correlation IDs

to a specific client request by filtering their logs based on this correlation identifier. By also incorporating timing data in this logging, one can use this technique to also retrieve performance data, such as the time spent on every operation. Figure 8.14 illustrates how correlation IDs can be used and an example of a trace that shows latency contribution of every application. There are several libraries and tools for implementing distributed tracing, such as OpenTracing[30] or Zipkin[31].

[30]See: https://opentracing.io
[31]See: https://zipkin.io

Chapter 9

Closing thoughts

Hopefully, this book has helped you understand how distributed systems can be useful, what are some of the main challenges one might face when building distributed systems and how to overcome them. Ideally, it has also made you realise that building or using a distributed system is a serious undertaking that should be done only when necessary. If your requirements around performance, scalability or availability can be met without the use of a distributed system, then not using a distributed system might actually be a wise decision.

This might feel like the end of a journey, but for some of you it might just be the beginning of it. For this reason, I think it would be useful to recap some key learnings from this book, while also highlighting topics that were left uncovered. In this way, those of you that are willing to dive deeper on some areas will have some starting points to do so.

First of all, we introduced some of the basic areas where distributed systems can help: performance, scalability and availability. Throughout the book, we analysed basic mechanisms that can help in these areas, such as partitioning and replication. It also became evident that these mechanisms introduce some tension between the aforementioned characteristics and other properties, such as consistency. This tension is formalised by basic theorems, such as the CAP theorem and the FLP result. This tension manifests in various ways, as shown in the book. For example, the decision on whether replication operates synchronously or asynchronously can be a trade-off between performance and availability or durability. Early on, we explained the difference between liveness and safety properties and tried to provide an overview of the basic consistency models that help formalise the behaviour of a distributed system and facilitate reasoning about its interaction with other systems. However, there are many more models we omitted in this book in the interest of time and simplicity, such as read-your-writes, monotonic reads and monotonic writes consistency models[87]. The first chapter also introduced the concept of failure detection and gave an example of a simplistic failure detector that makes use of heartbeats and timeouts. In reality, failure detectors have to deal with many more practical problems and need to be quite more complex. This might mean they have to avoid using timeouts in order to be applicable to quiescent algorithms[88], operate using a gossip protocol[1] to improve scalability and fault-tolerance[90] or output a suspicion level on a continuous scale instead of a binary value[91].

In the second chapter, we explored several partitioning techniques. Again, there are many more algorithms to study further if you want to, such as shuffle sharding[92] for workload and fault isolation, jump consistent hashing[93] for reduced memory overhead and better load distribution, multi-probe consistent hashing[94] and rendez-vous hashing[95]. The fourth chapter introduces the problem of consensus and explains the two major algorithms that solve it, Paxos and Raft. The topic of consensus is very old, but there is still a lot of very useful research being conducted in this area. For example, the original Paxos algorithm operated under the assumption that all quorums need to be majority quorums in order to maintain the safety properties of the algorithm. However, recent research[96] has actually demonstrated that this is not absolutely necessary. Instead, the algorithm just needs to ensure quorums from the first phase of the algorithm overlap with quorums from the second phase. This allows one to size the quorums of each phase accordingly depending on the requirements around performance and fault-tolerance during the steady state or during recovery. There are also a lot of variations of the Paxos algorithm, such as egalitarian Paxos[97] that does not require a stable leader, vertical Paxos[98] that allows a reconfiguration in the middle of a consensus round and more[99]. All these algorithms solve the consensus problem in cases where nodes can crash or get arbitrarily slow, but assuming they do not exhibit byzantine failures. Solving consensus under the presence of byzantine failures is a much more challenging task[100][101] and is subject to different constraints. There is also another category of algorithms worth exploring that solve the problem of consensus probabilistically[2], such as the

[1]Gossip protocols were also not covered in this book and they can be considered a whole topic on its own[89].

[2]This means that a value is agreed with a specific probability `p`, where `1-p` is the

one used in the Bitcoin protocol and known as the Nakamoto consensus[74].

In the fifth and sixth chapter, we introduced the notions of time and order and their relationship. We explained the difference between total and partial order, which is quite important in the field of distributed systems. While consensus can be considered as the problem of establishing total order amongst all events of a system, there are also systems that do not have a need for such strict requirements and can also operate successfully under a partial order. Vector clocks is one mechanism outlined in the book that allows a system to keep track of such a partial order that preserves causality relationships between events. However, there are more techniques that were not presented in the book. An example is conflict-free replicated data types (CRDTs)[102][103], which are data structures that can be replicated across multiple nodes, where the replicas can be updated independently and concurrently without coordination between the replicas and it's always possible to resolve inconsistencies that might result from this. Some examples of such data structures are a grow-only counter, a grow-only set or a linear sequence CRDT. The lack of need for coordination makes these data structures more efficient due to reduced contention and more tolerant to partitions, since the various replicas can keep operating independently. However, they require the underlying operations to have some specific characteristics (e.g. commutativity, associativity, idempotency etc.), which can limit their expressibility and practical application.

We believe it is a lot easier for someone to understand theory, when it is put into context by demonstrating how it is used in practical systems. This is the reason we included a chapter dedicated to case studies about real systems and how they use algorithms and techniques presented in the book. We tried to cover systems from as many different categories as possible, but we have to admit there are many more systems that are very interesting and we would like to include in this chapter, but we couldn't due to time constraints. The last chapter on practices and patterns was written under the same spirit and subject to similar time constraints. So, we call the reader to study more resources than those that were available in the book for a deeper understanding of how theory can be put in practice[104][105][106][107][108]. At the risk of being unfair to other systems and material out there, we would like to mention CockroachDB[3] as one system that has a lot of public material

probability of this value being considered not agreed (what is also referred to as reversed) in the future.

[3]See: https://github.com/cockroachdb/cockroach

demonstrating how they have used theoretical concepts in practice. Some concrete examples are how they implemented pipelined consensus[4] and how they implemented a parallelised version of two-phase commit[5] that required a single round-trip instead of two before acknowledging a commit. Some resources that also contain a lot of practical information on how to build and operate distributed systems are the Amazon Builders Library[6] and papers with learnings of practitioners that have built large-scale systems[109][110].

The last chapter included a discussion around high-level communication patterns. This chapter did not contain descriptions of basic protocols used widely, such as TCP, UDP, HTTP, DNS etc. This was done on purpose, since these protocols are quite involved and there are already many books containing detailed analyses about them. However, this does not imply these protocols are not as important. On the contrary, these protocols are extremely important and they can have significant implications on the behaviour of systems built on top of them. As a result, it is really important to have a good understanding of them when trying to build or use a system that makes use of them[7]. The last chapter also contained discussion about how systems can deal with failure. There are two types of failure that are frequently neglected when building or operating distributed systems even though they are quite common: gray failures[111] and partial failures[112]. Gray failures are those that do not manifest cleanly as a binary indication, but they are more subtle and they can be observed differently by different parts of a system. Partial failures are those in which only parts of a system fail in a way that has serious consequences equivalent to a full failure of the system sometimes due to a defect in the design. These types of failure can be very common in distributed systems due to many moving parts and they can have serious consequences, so it is very important for people that build and run distributed systems to internalise these concepts and look out for them in the systems they build and operate.

Finally, there are few topics that can be central or useful to how a distributed system operates, but they were not covered in this book either because they are a separate, extensive topic or because they are a bit more advanced. One such topic is security. In a distributed system, nodes are separated by network. Depending on the thread model in place, additional precautions

[4]See: https://www.cockroachlabs.com/blog/transaction-pipelining

[5]See: https://www.cockroachlabs.com/blog/parallel-commits

[6]See: https://aws.amazon.com/builders-library

[7]The analysis of the following incident is a good example of how such knowledge can be useful in mitigating and preventing issues: https://www.usenix.org/node/195676

might need to be taken in order to make sure the system operates in a secure way. The data transmitted in this network might need to be protected so that only authorised nodes can read it. The nodes might need to authenticate each other, so that they can be sure they are communicating with the right node and not some impersonator. There are a lot of cryptographic techniques that can help with these aspects, such as encryption, authentication and digital signatures. However, the interested reader will have to study them separately, since that's a separate field of study. We also did not examine how networks can be designed, so that distributed systems can run on top of them at scale, which is another broad and challenging topic[113]. Another important topic that we did not cover is formal verification of systems. There are many formal verification techniques and tools that can be used to prove safety and liveness properties of systems with TLA+[114] being one of the most commonly used across the software industry[115]. It is important to note that users of these formal verification methods have acknowledged publicly that they have not only helped them discover bugs in their designs, but they have also helped them significantly reason about the behaviour of their systems in a better way.

References

[1] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design (5th Edition).* 2011.

[2] A. B. Bondi, "Characteristics of scalability and their impact on performance," in *Proceedings of the second international workshop on Software and performance – WOSP '00. p. 195*, 2000.

[3] P. Bailis and K. Kingsbury, "The Network is Reliable," *ACM Queue, Volume 12, Issue 7, July 23, 2014*, 2014.

[4] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany, "An Analysis of Network-Partitioning Failures in Cloud Systems," *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, 2018.

[5] J. C. Corbett *et al.*, "Spanner: Google's Globally-Distributed Database," in *Proceedings of OSDI 2012*, 2012.

[6] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM. Volume 43 Issue 2, ACM. pp. 225–267*, 1996.

[7] F. Chang *et al.*, "Bigtable: A Distributed Storage System for Structured Data," in *Proceedings of 7th {USENIX} Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[8] G. DeCandia *et al.*, "Dynamo: Amazon's Highly Available Key-value Store," in *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, 2007.

[9] A. Lakshman and P. Malik, "Cassandra — A Decentralized Structured Storage System," *Operating Systems Review*, 2010.

[10] R. van Renesse and F. B. Schneider, "Chain Replication for Support-

ingHigh Throughput and Availability," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, 2004.

[11] D. K. Gifford, "Weighted voting for replicated data," in *Proceedings of the seventh ACM symposium on Operating systems principles*, 1979.

[12] N. Lynch and S. Gilbert, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *SIGACT News*, 2002.

[13] D. Abadi, "Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story," *Computer*, 2012.

[14] M. P. Herlihy and J. M. Wing, "Linearizability: A Correctness Condition for Concurrent Objects," *ACM Transactions on Programming Languages and Systems, July 1990*, 1990.

[15] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J. M. Hellerstein, and I. StoicaUC, "Highly Available Transactions: Virtues and Limitations (Extended Version)," *Proceedings of the VLDB Endowment*, 2013.

[16] ANSI X3.135-1992, "American National Standard for Information Systems - Database Language - SQL," 1992.

[17] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A Critique of ANSI SQL Isolation Levels," *SIGMOD Rec.*, 1995.

[18] C. H. Papadimitriou, "The Serializability of Concurrent Database Updates," *Journal of the ACM, October 1979*, 1979.

[19] P. A. Bernstein and N. Goodman, "Serializability theory for replicated databases," *Journal of Computer and System Sciences*, 1983.

[20] M. J. Franking, "Concurrency Control and Recovery," *SIGMOD '92*, 1992.

[21] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," *ACM Transactions on Database Systems, Volumne 6, Issue 2*, 1981.

[22] D. P. Reed, "Naming and Synchronisation in a Decentralised Computer System," *Tech. Rep. MIT/LCS/TR-204, Dept. Electrical Engineering and Computer Science, Massachusetts Institute of Technology*, 1978.

[23] A. Silberschatz, "A multi-version concurrency scheme with no rollbacks," *Proceedings of the first ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, 1982.

[24] R. E. Stearns and D. J. Rosenkratz, “Distributed database concurrency controls using before-values,” *Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, 1981.

[25] M. J. Cahill, U. Rohm, and A. D. Fekete, “Serializable Isolation for Snapshot Databases,” *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, 2008.

[26] D. R. K. Ports and K. Grittner, “Serializable Snapshot Isolation in PostgreSQL,” *Proceedings of the VLDB Endowment, Volume 5 Issue 12, August 2012*, 2012.

[27] T. S. and C. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-consistent Applications,” *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, 2014.

[28] J. N. Gray, “Notes on data base operating systems,” *Operating Systems. Lecture Notes in Computer Science, vol 60. Springer*, 1978.

[29] B. Lampson and H. E. Sturgis, “Crash Recovery in a Distributed Data Storage System,” 1979.

[30] H. Garcia-Molina, “Performance of Update Algorithms for Replicated Data in a Distributed Database.” 1979.

[31] D. Skeen, “Nonblocking Commit Protocols,” *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, 1981.

[32] D. Skeen, “A Quorum-Based Commit Protocol,” 1982.

[33] C. Binnig, S. Hildenbrand, F. Farber, D. Kossmann, J. Lee, and N. May, “Distributed Snapshot Isolation: Global Transactions Pay Globally, Local Transactions Pay Locally,” *The VLDB Journal, Volume 23 Issue 6, December 2014*, 2014.

[34] H. Garcia-Molina and K. Salem, “Sagas,” *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, 1987.

[35] L. Frank and T. U. Zahle, “Semantic ACID Properties in Multidatabases Using Remote Procedure Calls and Update Propagations,” *Software—Practice & Experience, Volume 28 Issue 1, Jan. 1998*, 1998.

[36] X. Defago, A. Schiper, and P. Urban, “Total order broadcast and multicast algorithms: Taxonomy and survey,” *ACM Computing Surveys,*

Volume 34, 2004.

[37] M. J. Fischer, N. A. Lynch, and M. S. Paterson, “Impossibility of Distributed Consensus with One Faulty Process,” *Journal of the ACM (JACM)*, 1985.

[38] L. Lamport, “The Part-time Parliament,” *ACM Transactions on Computer Systems (TOCS)*, 1998.

[39] L. Lamport, “Paxos Made Simple,” *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, 2001.

[40] T. D. Chandra, R. Griesemer, and J. Redstone, “Paxos Made Live: An Engineering Perspective,” *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, 2007.

[41] B. W. Lampson, “How to Build a Highly Available System Using Consensus,” *Proceedings of the 10th International Workshop on Distributed Algorithms*, 1996.

[42] H. Du, J. S. David, and Hilaire, “Multi-Paxos: An Implementation and Evaluation,” 2009.

[43] V. Hadzilacos, “On the Relationship between the Atomic Commitment and Consensus Problems,” *Fault-Tolerant Distributed Computing, November 1990, pages 201–208*, 1990.

[44] J. Gray and L. Lamport, “Consensus on Transaction Commit,” *ACM Transactions on Database Systems (TODS), Volume 31 Issue 1, March 2006*, 2006.

[45] D. Ongaro and J. Ousterhout, “In Search of an Understandable Consensus Algorithm,” *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, 2014.

[46] C. Dyreson, “Physical Clock,” *Encyclopedia of Database Systems*, 2009.

[47] M. Raynal and M. Singhal, “Logical time: capturing causality in distributed systems,” *Computer, Volume 29, Issue 2, Feb 1996*, 1996.

[48] L. Lamport, “Time, Clocks, and the Ordering of Events in a Distributed System,” *Communications of the ACM 21, 7 July 1978*, 1978.

[49] S. Reinhard and M. Friedemann, “Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail,” *Distributed Computing, Volume 7 Issue 3, March 1994*, 1994.

[50] F. Colin J., “Timestamps in Message-Passing Systems That Preserve the Partial Ordering,” *Proceedings of the 11th Australian Computer Science Conference (ACSC’88), pp. 56–66*, 1988.

[51] M. Friedemann, “Virtual Time and Global States of Distributed Systems,” *Parallel and Distributed Algorithms*, 1988.

[52] B. Charron-Bost, “Concerning the Size of Logical Clocks in Distributed Systems,” *Information Processing Letters, Volume 39 Issue 1, July 12, 1991*, 1991.

[53] D. S. Parker *et al.*, “Detection of mutual inconsistency in distributed systems,” *IEEE Transactions on Software Engineering, Volume, 9 Issue 3, pages 240-247*, 1983.

[54] N. M. Perguica, C. Baquero, P. S. Almeida, V. Fonte, and G. Ricardo, “Dotted Version Vectors: Logical Clocks for Optimistic Replication,” *arXiv:1011.5808*, 2010.

[55] K. M. Chandy and L. Lamport, “Distributed Snapshots: Determining Global States of Distributed Systems,” *ACM Transactions on Computer Systems (TOCS), Volume 3 Issue 1, Feb. 1985*, 1985.

[56] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google File System,” *Sumposium on Operating Systems Principles ’03*, 2003.

[57] K. Shvachko, H. Kuang, and R. Chansler, “The Hadoop Distributed File System,” *IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010.

[58] M. Burrows, “The Chubby Lock Service for Loosely-coupled Distributed Systems,” *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, 2006.

[59] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, “ZooKeeper: Wait-free Coordination for Internet-scale Systems,” *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, 2010.

[60] F. P. Junqueira, B. Reed, and M. Serafini, “Zab: High-performance Broadcast for Primary-backup Systems,” *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks*, 2011.

[61] A. Medeiros, “ZooKeeper’ s atomic broadcast protocol : Theory and practice,” 2012.

[62] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica, Volume 33 Issue 4, 1996*, 1996.

[63] D. F. Bacon *et al.*, "Spanner: Google's Globally-Distributed Database," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017.

[64] E. Brewer, "Spanner, TrueTime and the CAP Theorem," 2017.

[65] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis,II, "System Level Concurrency Control for Distributed Database Systems," *ACM Transactions on Database Systems (TODS), volume 3, Issue 2, June 1978*, 1978.

[66] A. Thomson, T. Diamond, S.-C. Weng, K. Ren, P. Shao, and D. J. Abadi, "Calvin: Fast Distributed Transactions for Partitioned Database Systems," *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, 2012.

[67] D. J. Abadi and J. M. Faleiro, "An Overview of Deterministic Database Systems," *Communications of the ACM, Volume 61 Issue 9, September 2018*, 2018.

[68] J. Kreps, N. Narkhede, and J. Rao, "Kafka : a Distributed Messaging System for Log Processing," *NetDB' 11, June 2012, 2011*, 2011.

[69] G. Wang *et al.*, "Building a Replicated Logging System with Apache Kafka," *NetDB' 11, June 2012, 2011*, 2011.

[70] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," *Proceedings of the European Conference on Computer Systems, Eurosys*, 2015.

[71] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *ACM Queue, volume 14, pages 70-93, 2016*, 2016.

[72] R. G. Brown, "The Corda Platform: An Introduction," 2018.

[73] M. Hearn and R. G. Brown, "Corda: A distributed ledger," *August 2019, version 1.0*, 2019.

[74] S. Nakamoto, "Bitcoin: A Peer-to-Peer Electronic Cash System," 2008.

[75] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation, Volume 6*, 2004.

[76] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, 2010.

[77] M. Zaharia *et al.*, "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing," *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2012.

[78] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas, "Apache Flink™: Stream and Batch Processing in a Single Engine," *IEEE Data Engineering Bulletin 2015*, 2015.

[79] P. Carbone, S. Ewen, G. Fora, S. Haridi, S. Richter, and K. Tzoumas, "State Management in Apache Flink: Consistent Stateful Distributed Stream Processing," *Proceedings of the VLDB Endowment, Volume 10 Issue 12, August 2017*, 2017.

[80] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized Streams: Fault-tolerant Streaming Computation at Scale," *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, 2013.

[81] T. Akidau *et al.*, "MillWheel: Fault-tolerant Stream Processing at Internet Scale," *Proceedings of the VLDB Endowment, Volume 6 Issue 11, August 2013*, 2013.

[82] T. Akidau *et al.*, "The Dataflow Model: A Practical Approach to Balancing Correctness, Latency, and Cost in Massive-scale, Unbounded, Out-of-order Data Processing," *Proceedings of the 41st International Conference on Very Large Data Bases, Volume 8 Issue 12, August 2015*, 2015.

[83] P. Carbone, G. Fora, S. Ewen, S. Haridi, and K. Tzoumas, "Lightweight Asynchronous Snapshots for Distributed Dataflows," 2015.

[84] C. G. Gray and D. R. Cheriton, "Leases: An Efficient Fault-Tolerant Mechanism for Distributed File Cache Consistency," *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, 1989.

[85] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-End Arguments in System Design," *ACM Transactions in Computer Systems 2, 4, November, 1984, pages 277-288*, 1984.

[86] T. Moors, "A critical review of End-to-end arguments in system design," *Communications, ICC 2002. IEEE International Conference on, Volume 2,*

2002.

[87] P. Viotti and M. Vukoliundefined, “Consistency in Non-Transactional Distributed Storage Systems,” *ACM Computing Surveys, Volume 49, No. 1*, 2016.

[88] A. M. Kawazoe, W. Chen, and S. Toueg, “Heartbeat: A timeout-free failure detector for quiescent reliable communication,” *11th International Workshop on Distributed Algorithms, ’97*, 1997.

[89] K. Birman, “The Promise, and Limitations, of Gossip Protocols,” *ACM SIGOPS Operating Systems Review, October 2007*, 2007.

[90] R. van Renesse, Y. Minsky, and M. Hayden, “A Gossip-Style Failure Detection Service,” *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, ’98*, 1998.

[91] N. Hayashibara, X. Défago, R. Yared, and T. Katayama, “The phi accrual failure detector,” *23rd IEEE International Symposium on Reliable Distributed Systems*, 2004.

[92] C. MacCárthaigh, “Workload isolation using shuffle-sharding,” *The Amazon Builders’ Library*, 2019.

[93] J. Lamping and E. Veach, “A Fast, Minimal Memory, Consistent Hash Algorithm,” *arXiv:1406.2294*, 2014.

[94] B. Appleton and M. O’Reilly, “Multi-probe consistent hashing,” *arXiv:1505.00062*, 2015.

[95] D. G. Thaler and C. V. Ravishankar, “A Name-Based Mapping Scheme for Rendezvous,” *University of Michigan Technical Report CSE-TR-316-96*, 2013.

[96] H. Howard, D. Malkhi, and A. Spiegelman, “Flexible Paxos: Quorum intersection revisited,” *arXiv:1608.06696*, 2016.

[97] I. Moraru, D. G. Andersen, and M. Kaminsky, “There is More Consensus in Egalitarian Parliaments,” *Proceedings of the 24th ACM Symposium on Operating Systems Principles*, 2013.

[98] L. Lamport, D. Malkhi, and L. Zhou, “Vertical Paxos and Primary-Backup Replication,” *Proceedings of the 28th Annual ACM Symposium on Principles of Distributed Computing*, 2009.

[99] R. Van Renesse and D. Altinbuken, "Paxos Made Moderately Complex," *ACM Computing Surveys, February 2015, Article No. 42*, 2015.

[100] M. Castro and B. Liskov, "Practical Byzantine Fault Tolerance," *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, 1999.

[101] L. Lamport, "Byzantizing paxos by refinement," *Proceedings of the 25th International Conference on Distributed Computing*, 2011.

[102] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free Replicated Data Types," *[Research Report] RR-7687, 2011, pp.18. inria-00609399v1*, 2011.

[103] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of Convergent and CommutativeReplicated Data Types," *Research Report] RR-7506, Inria – Centre Paris-Rocquencourt; INRIA. 2011, pp.50. inria-00555588*, 2011.

[104] R. Nishtala *et al.*, "Scaling Memcache at Facebook," *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, 2001.

[105] N. Bronson *et al.*, "TAO: Facebook's Distributed Data Store for the Social Graph," *Proceedings of the 2013 USENIX Conference on Annual Technical Conference*, 2013.

[106] B. H. Sigelman *et al.*, "TAO: Facebook's Distributed Data Store for the Social Graph," 2010.

[107] A. Verbitski *et al.*, "Amazon Aurora: On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes," *Proceedings of the 2018 International Conference on Management of Data*, 2018.

[108] A. Verbitski *et al.*, "Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases," *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017.

[109] J. Hamilton, "On Designing and Deploying Internet-Scale Services," *Proceedings of the 21st Large Installation System Administration Conference (LISA '07)*, 2007.

[110] E. A. Brewer, "Lessons from Giant-Scale Services," *IEEE Internet Computing, Volume 5, No. 4*, 2001.

[111] P. Huang *et al.*, “Gray Failure: The Achilles’ Heel of Cloud-Scale Systems,” *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, 2017.

[112] C. Lou, P. Huang, and S. Smith, “Understanding, Detecting and Localizing Partial Failures in Large System Software,” *17th USENIX Symposium on Networked Systems Design and Implementation*, 2020.

[113] B. Lebiednik, A. Mangal, and N. Tiwari, “Understanding, Detecting and Localizing Partial Failures in Large System Software,” *arXiv:1605.01701*, 2016.

[114] M. A. Kuppe, L. Lamport, and D. Ricketts, “The TLA+ Toolbox,” *arXiv:1912.10633*, 2019.

[115] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, “How Amazon Web Services Uses Formal Methods,” *Communications of the ACM, Volume 58, No. 4*, 2015.