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 3$^{\text{rd}}$ 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.

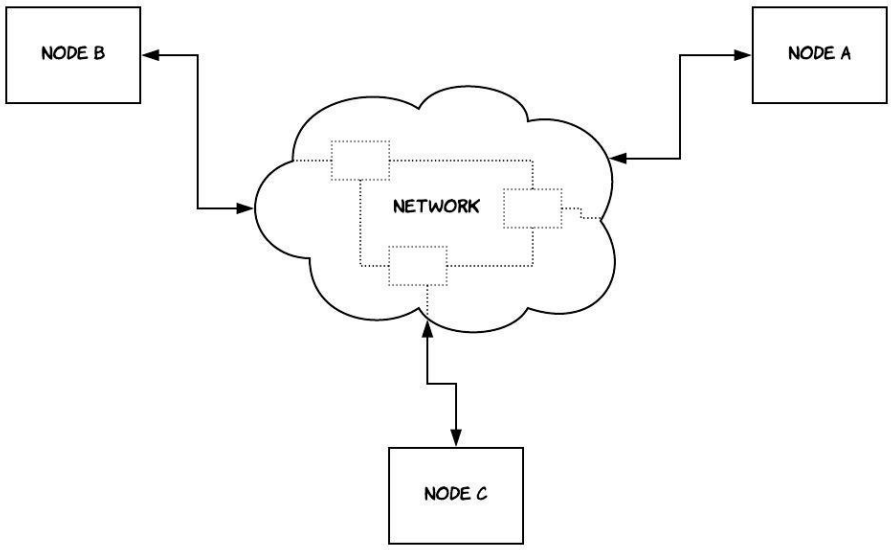

Figure 1.1: A distributed system

- 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.

As you progress through the book, you will get a deeper understanding of why these statements are fallacies. However, we will try and give you a sneak preview here by going quickly over them and explain where they fall short. The first fallacy is sometimes enforced by abstractions provided to developers from various technologies and protocols. As we will see in a later chapter networking protocols, such as TCP, can make us believe that the network is reliable and never fails, but this is just an illusion and can have significant repercussions. Network connections are also built on top of hardware that will also fail at some point and we should design our systems

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

accordingly. The second assumption is also enforced nowadays by libraries, which attempt to model remote procedure calls as local calls, 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. The fourth fallacy illustrates the fact that the wider network that is used by two nodes in order to communicate is not necessarily under their control and thus should be considered insecure. The book contains a chapter dedicated on security that explains various techniques that can be used in order to make use of this insecure network in a secure way. This network is also composed of many different parts that might be managed by different organisations with potentially different hardware and failures in some parts of this network might mean its topology might have to change to remain functional. This is all highlighted by the fifth, sixth and eighth fallacies. Last but not least, transportation of data between two points incurs financial costs, which should be factored in when building a distributed system.

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

[3] See: https://grpc.io/
[4] See: https://thrift.apache.org/

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 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. This is in contrast to memory operations that can provide much stricter guarantees[6]. For instance, in a distributed system messages might

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

take extremely long to be delivered, they might be delivered out of order or not at all.

Partial failures are cases where only some components of a distributed system fail. This behaviour can come in contrast to certain kinds 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 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.

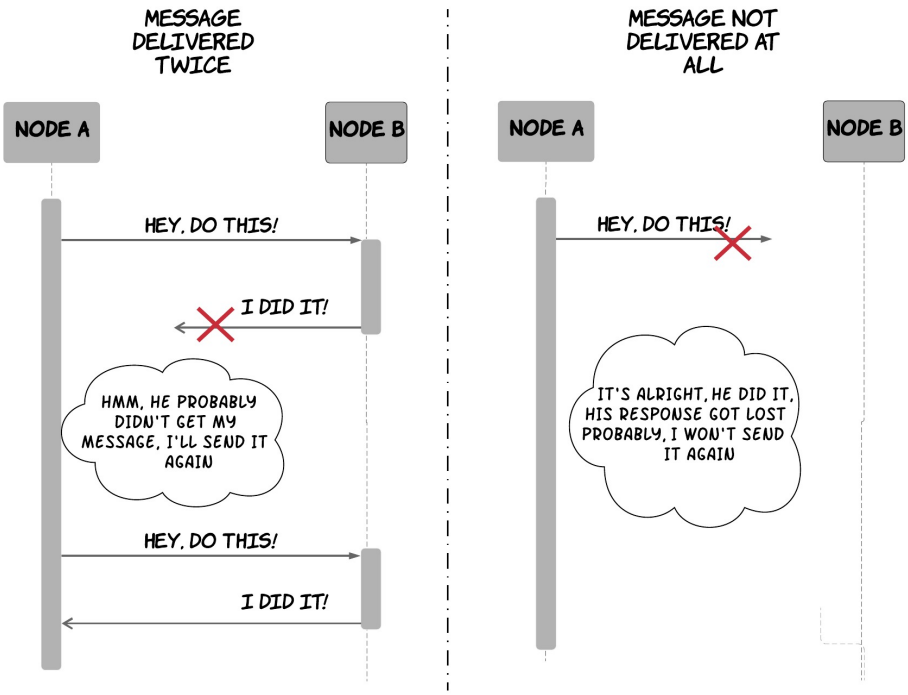

Figure 1.2: Intricacies of a non-reliable network in distributed systems

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 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.

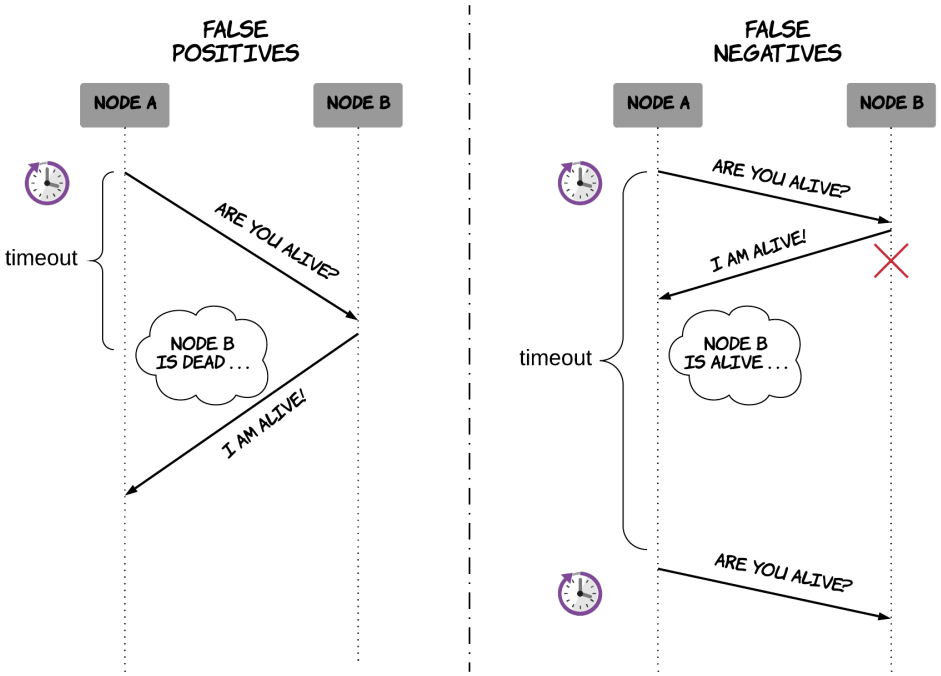

Figure 1.3: Trade-offs in failure detection

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 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 stateless components (which are performing business capabilities) and stateful 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 the most challenging problems we will cover in this book are present mostly in stateful systems.

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.