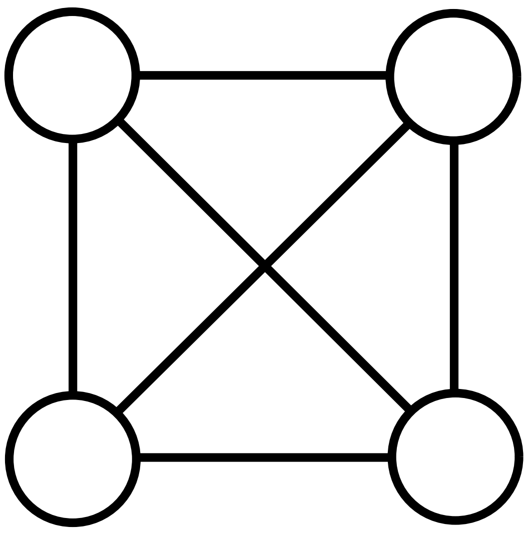
Secure Distributed Systems

Ivan Damgård
Jesper Buus Nielsen
Claudio Orlandi



August 23, 2019

Contents

Preface

This is the second edition of a book on security and distributed systems. It was first written in Spring of 2018 to be used in Fall 2018 in the first edition of the course Security and Distributed Systems at Aarhus university. It was rewritten in Spring 2019 based on the input from the first version of the course. The parts on security are to some extend reused from a previous course on security, but have been rewritten. The parts on distributed systems have been written from scratch, but are based on experiences from teaching a course on distributed system for several years.

In the security part of the book, we will take a "bottom-up" approach: we first look at some technical solutions that are often part of the design of security mechanisms, including things like cryptography, network security protocols, access control, etc. Then we look at some higher level concepts, such as the security policies discussed above. Finally, we take a look at some of the reasons why systems fail in real life – in the language from this section, cases where either the threat model failed to capture the actual attacks, or where the security mechanisms failed to enforce the policy.

In the distributed systems part of the book, we have chosen to focus mainly on the so-called Byzantine agreement problem. Distributed systems is a huge area, and any attempt to cover the whole ground would lead to a superficial treatment. Thinking correctly about the security of distributed systems requires a change of mindset, which in our experience is not achieved by a superficial treatment. It is necessary to get into some of the technical details of some systems and understand why they are designed as they are. Decades of experience in teaching distributed systems has revealed that studying the Byzantine agreement problem is a good choice, as it crystallises the hard problems of secure distributed system into a problem which is easy to study. Our hope is that the insights gained by studying the Byzantine agreement problem will give a change of mindset which will allow the reader to understand, design, and implement many other secure distributed systems.

Acknowledgements

We would like to thank the following people for helpful feedback on earlier versions of the book: Rebekah Hannah Mercer, Nikolaj Sidorenco, Mark Jensen, Kira Kutscher, Matias Frank Jensen, Mark Simkin, Adam Blatchley Hansen, ...

1

Introduction

Contents

1.1 IT-Security

1.1.1 Some basic concepts

The subject of security is extremely difficult to define in a precise way. However, most people seem to agree that when it comes to security of IT systems, there are three fundamental aspects, sometimes referred to as *security objectives*:

Confidentiality: we want systems where information does not leak to people that should not have access. This applies at any time, i.e., whether the data is being stored, processed or transmitted.

Authenticity: we want the data in our systems to be authentic, i.e., that it has not been tampered with by people not authorized to do so. When this applies to the actual data, this property is sometimes called *data* *integrity*. More broadly, authenticity can also refer to beliefs about the data: common examples include the identity of the sender, the date of creation, etc. For instance, authenticity might mean that a user should not be fooled into believing that a document comes from a particular source when this is not the case.

Availability: we want systems that work when they are supposed to, i.e., legal users of the system should be able to get their data when they need them.

These objectives are just intended to be loose descriptions of basic user demands on a system. They are not linked to particular solutions, in fact there are many technically different ways to reach the goals, offering different levels of security. It should also be noted that if IT security is a fuzzy concept, the terminology of the field is even fuzzier, so you may well find other texts where the terms defined here are not used in exactly the same way. This appears to be something one simply has to live with. In this book, however, we will try to be consistent.

It is also useful to point out one thing that IT-security does *not* cover: the general issue of reliability or correctness of programs will not be our main concern in the security part of the book. This is not to say that a program failing to perform according to specification cannot be a security concern. Indeed, some of the worst security problems in practice occur because of bugs in software, and we will certainly look at such issues. The point is, however, that correctness and reliability *in general* would be relevant problems even in a world of perfect gentlemen where no one was attacking security, and so correctness and reliability are not as such security issues. Conversely, not all security problems come from software bugs, so even if programmers never made mistakes, we would still have to worry about attacks on security.

1.1.2 Security Policies

For any realistic situation, one needs a description of what security objectives we are after. This needs to be much more precise than just talking about confidentiality, authenticity and availability, and it is important to realize that this comes first, i.e., we have to define the objectives before we start thinking about how we are going to reach them. Such a description is called a *security policy*. In a formal, abstract model, this can take the form of specifying that certain states of the system are unsafe and should be avoided, while the others are defined to be secure. In a programming language environment, such as Java, it can take the form of a set of rules for what different programs or software components are allowed to do, what data they can access, etc. When systems involve human beings as active participants, the security policy is often much less precise, and is formulated in a language that makes immediate sense to humans. While this is good from the point of view of having people understand what the goal is, the danger is that different people interpret the policy in different ways, leading to a multitude of security problems in practice.

Summarizing, a security policy, whatever form it has, is basically an attempt to characterize the events we do not want to see in a system, separating them from the allowed behavior.

Depending on whom you talk to, or which book you read, a security policy may include more than this, namely it does not only describe security objectives, it may also describe some strategies or procedures aimed at ensuring that the security objectives are reached. For instance, a company may have a security policy saying:

"This policy aims to ensure that virus attacks on our machines are prevented. All machines we use in the company must therefore use virus protection software. A single person should be made responsible for every machine, and this person must check at least every month that the virus scanner is updated".

This somewhat blurs the distinction between security policies and other concepts in the area. But usually, high-level procedures such as the one above are thought of as part of the security policy, while lower level technical solutions are referred to security mechanisms (see below).

Let us finally emphasize the importance of being as concrete and precise as possible, even when phrasing high-level security policies in human language. Consider the following example:

"This company takes security very seriously. We realize that virus attacks are a serious problem and we will take every possible measure to prevent them"

This is a political statement and not a security policy! Notice the differences between this and the above. Even though the texts are so short, some basic properties of a proper security policy can be identified: it defines reasonably precisely the system we are talking about and what the goal is.

1.1.3 Threat Model

Any system in real life can be *attacked*. An attacker can be an honest but careless user who, for instance, happens to release a piece of secret data. The attacker can be a malicious user, such as a bank employee who reprograms the system to send money to his own account. An attacker can be an intruder, such as a hacker who breaks in and takes control over a machine. And finally, the attack can come from nature itself, for instance when machines malfunction, causing loss of data.

It is almost never feasible to design a system secure against *any* attack, or if it is, the resulting system is too expensive or too slow. We therefore need a *Threat model*, that defines which attacks we are going to worry about. Whether a given system is secure or not can depend dramatically on which threat model is considered. Imagine, for instance, a multi-user computer system that stores and handles passwords internally with a very high level of security. However, the system administrator keeps a note with the administrator password on his desk. If the threat model considers only attacks that users can carry out from their own workstation, the system may be perfectly fine (and this may be reasonable if there is a high degree of physical security in the building, so that only the administrator can enter his own office). But if the threat model includes attacks where you gain undetected entrance to the administrator's office (sometimes known as *the evil maid attack*), the system is totally insecure.

1.1.4 Security Mechanisms

Given a threat model and security policy, in order to ensure that the behavior of a system follows the security policy, when subjected to the attacks identified in the threat model, we need certain *security mechanisms.* These are designed to (hopefully) prevent attackers from forcing unwanted events to happen. A security mechanism can use any technique that will help us reach the goal: physical locks, ID cards, password protection, virus checkers, cryptography, etc.

For instance, if the security policy of a hospital says that only doctors should have access to the medical records of patients, encrypting the hospital's patient database can be part of a security mechanism enforcing this policy. Another part may be a password-based scheme to ensure that only users registered as doctors can make the system decrypt data from the database.

1.1.5 Is it secure?

We have now arrived at an important question: given that we have defined a security policy, a threat model and that we have designed security mechanisms to enforce it: will the real system actually behave according to the policy? Or, put more directly, can the system be successfully attacked? This is perhaps *the* most important question in IT security, and unfortunately also the most difficult question to answer. Many attempts have been made to give answers in various contexts that are at least better than "I believe it's secure". But there are some basic reasons why we never seem to reach the final goal of giving definite answers as to whether a given system is secure:

To prove something about a concrete system, one would usually try to make a mathematical model of it and try to prove a theorem stating that the system never shows a behavior we do not want. Of course, such a model must also include one or more attackers behaving according to the threat model. Now, you may have one of two problems:

- The threat model is so general that we capture (almost) all possible attacks and behaviors. In this case, it is generally speaking very hard or impossible to prove anything. In several cases, it is even known that we will be up against an undecidable problem, i.e., it is known that we *cannot* prove security.
- The threat model is restricted sufficiently so that we can indeed prove that the system is secure. But then, of course, we do not know what happens if the attacker finds a strategy that is not captured in the model. Put differently: a very restricted threat model is less likely to reflect the risks that are really out there.

This is of course an unfortunate state of affairs. However, in this course, we will take a practical point of view: first, security is clearly needed in real life, and the fact that the theory is incomplete is no excuse for doing nothing about security. Second, the fact that the theory is incomplete does not mean it is useless! A proof of security in a restricted threat model can still tell us that a certain class of potential security holes have already been closed, so that we can focus our attention on other attacks.

1.2 Cryptography

This section gives a very informal introduction to some basic concepts from cryptography.

It should be apparent from the introduction that cryptography is not the only technique that is needed to obtain secure systems. In fact one can argue that cryptography can *never* be the complete answer to security problems – we return to this later. Nevertheless, cryptography is often an essential part of the security mechanisms we need.

Basically, the entire crypto-world can be divided in two orthogonal ways. The first type of division relates to the *kind of problem* we are trying to solve. The second kind relates to the type of security we obtain.

1.2.1 Confidentiality versus Authenticity

The security objective we are after with a cryptographic solution may be either *confidentiality* or *authenticity*, objectives that were already described in the overview above. Cryptology is most often not useful in trying to ensure availability, since this goal is much more dependent on the way data is handled and stored physically. We will discuss availability in more detail in the chapters on distributed systems, for instance when we cover replication and the so-called CAP theorem.

It is important to understand that the two kinds of security objectives really are different and call for different technical solutions. Simply encrypting all the data you send is *not* always the solution, and is in some cases even counter-productive!

Thus, in a given real-life scenario, you *must always* find out first whether you are dealing with an authenticity or a confidentiality problem (or both), and then choose the appropriate cryptographic technique for solving the problem.

1.2.2 Unconditional versus Computational security

The second type of division relates to the *kind of security provided.* Cryptographic constructions may provide *unconditional* or only *computational* security. For instance, a method for encrypting data may offer such strong protection against eavesdropping that even a hacker with unlimited computing power will not be able to break the encryption. Or it may be the case that the system could in principle be broken, but only by spending a completely unrealistic amount of computing power (and hence also a completely unrealistic amount of real time). In the latter case, we talk about computational security.

The unconditionally secure systems are generally speaking not very interesting from a practical point of view, at least when “practical” is to be “commercially interesting”. We know this due to some basic theory which tells us that to provide – for instance – unconditional confidentiality, we will need to either supply both sender and receiver with very long strings of secret bits, or rely on natural noise occurring on communication channels, or rely on non-standard ways of communication, such as quantum communication. With current state of the art, this is either expensive or hard to control in practice. So we concentrate in the following on computational security.

The world of computational security can be further subdivided, namely into *secret-key* and *public-key* systems. The meaning of this will hopefully become clear in the following.

1.3 Distributed Systems

1.3.1 What is a Distributed System?

A distributed system is a collection of computers which collaborate to solve a joint problem and try to appear as one consistent system towards the user or application layer. In this book we only concern ourselves with distributed systems with the following properties:

Geographic separation: The machines in the system are distributed over a large geographic area. They are typically also located in different administrative domains with different system administrators.

No common physical clock: The machines do not share a physical clock, so they cannot coordinate via some notion of global time. They might have local physical clocks, but these are near impossible to synchronise perfectly and tend to drift apart if nothing is done to synchronise them. This is an issue we discuss in more detail in Chapter 8.

Coordination is via message passing: The machines coordinate via passing messages. In this note, we will typically assume that communication is via TCP/IP. Because of the geographic separation it is hard to give an estimate on how long it takes a messages to be delivered. For that reason it can be hard to distinguish a slow message from a message that was lost or never sent.

Heterogeneous: The machines have different hardware, different operating system and different computational power and connectivity. Some machines might be powerful servers in a server park and other might be laptops, cell phones and even cheap sensors with limited network and battery time.

An example is the Dropbox system which consists of about $10,000$ servers used by Dropbox and all the machines of the users.

Some computing systems falling outside the above class, but which still have flavours of distribution are multicore computers and cluster computing. Modern computers typically have many cores. They communicate via shared memory and share a physical clock. A step up from this are cluster computers which are designed for parallel execution of computationally intense tasks. In this case, a large number of homogeneous computers are connected via a high speed network and controlled by a master node. These machines share one operating system and administrator. The main problems encountered in these settings are best covered in a course on parallel computing.

A distributed system is often provided via a *middleware* layer running on top of the different operating systems at the physically separated machines. The main goals of such middleware are:

API defined: The middleware provides some uniform API to the application on all the different machines using the distributed system. The behaviour of the system is ideally defined only via this API (e.g., on which inputs does it give which outputs). This means that the implementation can be replaced with any other version that satisfies the API without the application layer noticing that part of the system has been changed.

Transparency: Ideally the middleware hides the quirks of the different operating systems and the fact that the system is distributed. For instance, the Dropbox system allows you to access the same files across several types of computers and cell phones and gives you the illusion that the same files are present on all your devices (despite the fact that e.g., different systems store files in very different ways!).

Openness/Interoperability: The system should ideally be defined via standards or interfaces such that different operators can implement their own versions. An example is the SMTP e-mail which can be used to exchange email across a wide range of operating systems.

1.3.2 Why Distributed Systems?

Some of the main reasons for making distributed systems are:

Resource sharing: Often resources are inherently distributed and shared, like printers, large central databases or people who need to collaborate despite being geographically separated.

Fault tolerance: Having more machines gives the possibility that the system might keep operating if some of the are faulty. We might for instance replicate a resource like a file to make sure it is available even if one of the machines it was stored on has crashed. As an example Dropbox is replicating your files such that if any single server crashes no files are lost or become unavailable. This is an example of ensuring availability using replication.

Scalability / Load balancing: Having more machines that share the work can make the system faster. You can for instance store a very popular file on

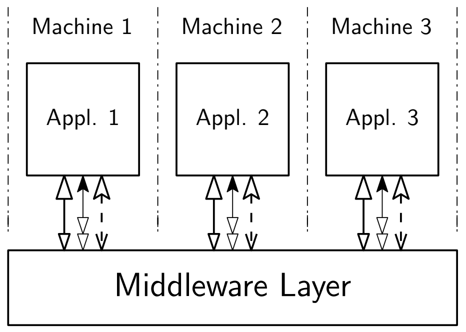


Figure 1.1 An illustration of how a good middleware layer should look like from the viewpoint of applications running on different machines. They all have some well-defined API to the system. The system looks like one coherent system to the applications. No details about the local operating system or how the middleware layer is implemented on the local machine is visible to the applications.

100 different servers evenly distributed across the planet. If you implement a distributed system which allows users to locate the closest server and retrieve the file from there, each server will get much less workload and clients will get shorter access time.

Efficiency: It is often much more economically viable to implement a given task using a distributed system of cheap components than by constructing a single super-powerful or super-reliable machine which handles the task alone.

Autonomy: You might want your system to be autonomous in the sense that it is not controlable by a single person or organisation. This is the case for peer-to-peer filesharing systems and the Bitcoin system which we will cover in later chapters. An autonomous system of course cannot run on a single physical machine. Otherwise, the owner of the machine could take over the system, or just cut the power cord. Neither can an autonomous system run inside a single organisation. Nor should the system run solely inside a single country, which might shut it down using legislation. To ensure that no entity could take over the system, it is necessary to set up a system where anyone can join and help run the system and where there is a large number of participants from all over the planet.

Example 1.1 (Replication for efficiency) Consider the following toy example for how to use replication for efficiency. Assume you can buy a cheap but crappy server that crashes on any given day with probability $\frac{1}{1{,}000}$. Assume the probability it crashes on a given day is independent of whether it crashes on any other day. Then the probability it crashes in a given year is about 31%. Assume that by buying a machine that costs 10 times as much you get a machine that crashes

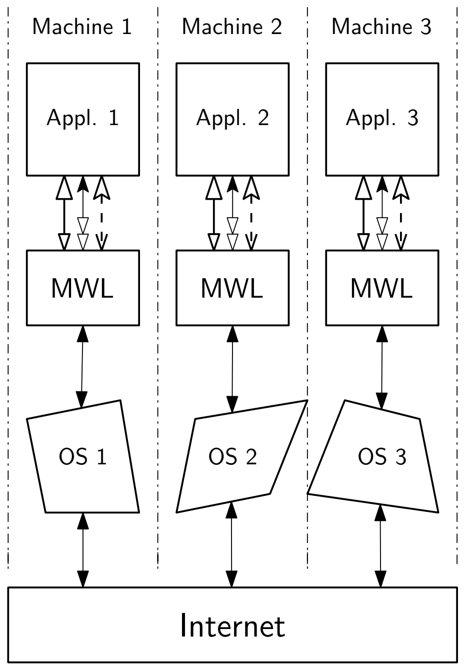


Figure 1.2 An illustration of how a middleware layer is actually implemented. On each machine there is a middleware application running, called MWL here, which via the local operating system and ultimately the Internet communicates with the MWL application on other machines.

on a given individual day with probability only $\frac{1}{100{,}000}$. The probability it crashes in a given year is now about 0.4%. Here is a trick that will both save you money and give you higher reliability. Assume that we can make a distributed system consisting of three machines with the following characteristics: If at most one machine is crashed, then the system is still operational. If within a given day at most one machine crashes, then we have time to replace it with a fresh machine. Assuming machines crash independently, the only way the system can crash is that two machines crash on the same day. If you use three crappy machines, the probability this happens in a given year is about 0.1%. So we enhanced the availability of the system to be better than that of the expensive machine and only paid three times the price of a crappy machine.

$\triangle$

1.3.3 Replication and Consistency

As we will see later, one of the main challenges in distributed systems is:

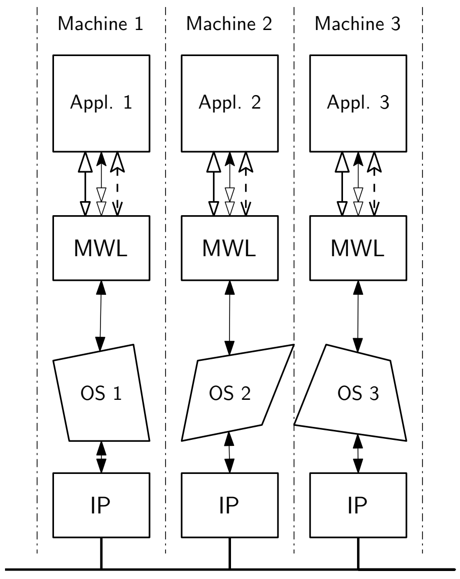


Figure 1.3 Of course the internet itself is an abstraction like a middleware layer. Via the TCP/IP layer it makes the internet look "the same" to the MWL programs on different operating systems. Each OS runs its own implementation which via a series of physical links talk to the TCP/IP implementation in other operatings systems on other physical machines.

Consistency: When a resource is replicated in a distributed system it should be kept consistent across all replicas. For instance, if you make a change to a file on one location and later read the file on some other location, then you expect to see the change that you made.

Consistency is often hard even to define and even harder to implement. If a file is replicated across several servers, we cannot simply require that it is always the same on all replicas. A change will take time to propagate across the network. And even if we tried to make all replicas apply the change at the same physical time this cannot be done with perfect precision. One of the files will be written first and for a moment the system is inconsistent. Consistency is therefore more subtle to define. It is defined via a contract between the clients and the middleware. If the clients promise to behave in a particular way, then the values they receive from the middleware will be consistent according to some definition. Consistency is defined via the input-output behaviour of the distributed system and not via the particular internal representation. The specification is done in two steps. First the syntax is given, specifying which inputs and outputs the system can receive and give. Then the required relation between the inputs and outputs is specified. The last step can either be done by specifying an ideal system which has the

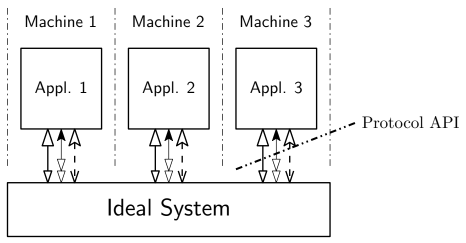


Figure 1.4 When defining what a system is supposed to do, we specify an ideal system which has the same API as the real system. It is programmed as a single object. We program it such that on any given inputs it gives the right outputs, in the right places at the right time. We think of it as a single object magically present at all sites at ones. Therefore the implementation does not have to take care of things like message passing or crashing machines. In implementing it one only focuses on making sure the input-output behaviour is as intended. It will for instance keep only one copy of a file and present that file to all users at all sites, maybe with some allowed delay. Since the ideal system is a specification of the intended behaviour, it is consistent by definition.

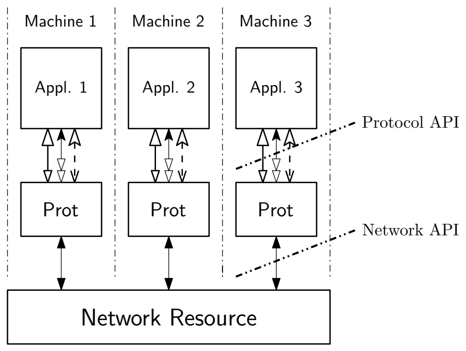


Figure 1.5 In the real system each machine has its own program and must communicate via some network resource, like the Internet. Here each machine might have its own copy of a file and try to keep them consistent via message passing over the network resource. One compares the real system to the ideal system by saying that any input-output behaviour of the real system is also possible by the ideal system. This argues that the real system does not doing anything which is disallowed.

intended input-output behaviour, or just be giving the intended relation between the inputs.

Example 1.2 (Consistency in a replicated system) As an example of how to define and keep consistency consider the following system with three servers that try to store a file for a single client. Initially the file is set to be the empty string ϵ. The system has a client which is the one that gets inputs and gives outputs.

First the syntax:

Read On input (read) the client eventually outputs (read, f).

Change On input (change, C) for some function C the client eventually outputs $(\mathsf{changed})$.

The intended meaning of the read command is that the returned f is the current value of the file. The intended behaviour of the change command is that C is applied to the current value of the file. We will now define this a bit more precisely.

First say that the input to the clients are legal if the next input is always given after the previous input is “returned”. For instance after having given input (read), you are not allowed to input (read) to the client again until after it outputs (read, f). We can now define consistency as follows. Consider any legal sequence of inputs to the client and let $C_1, \ldots, C_n$ be all the change functions in this sequence in the order in which they appear. If the last input to the client is (read) and the client eventually outputs (read, f), then $f = C^n(\epsilon)$, where:

$$C^n = C_n \circ \cdots \circ C_2 \circ C_1 \ .$$

Let us attempt a distributed implementation. We will use three servers. Each server stores a pair (t, f), where t is a timestamp (which is simply an integer initially set to 0), and f is the file (initially set to the empty string). If at any point in time the values stored by the servers are $(t_1, f_1), (t_2, f_2), (t_3, f_3)$, then we define m to be the smallest $m \in \{1, 2, 3\}$ such that $t_m = \max\{t_1, t_2, t_3\}$. We call t_m the current timestamp and we call f_m the current file. Intuitively we just say that the current version of the file is the one with the highest timestamp-

To make a system that keeps working even if one server is unresponsive, we make a system where the client can read or modify the file with access to only two servers. Here is how we implement the system:

Read On input (read) the client reads from any two servers that are available. Call the received values (t_1, f_1) and (t_2, f_2). If $t_1 \geq t_2$ it outputs (read, f_1). Otherwise it outputs (read, f_2).

Change On input (change, C) for some function C the client reads from any two servers that are available. Call the received values (t_1, f_1) and (t_2, f_2). If $t_1 \geq t_2$ it lets $(t, f) = (t_1, f_1)$. Otherwise it lets $(t, f) = (t_2, f_2)$. Then it sets $t' = t + 1$ and $f' = C(f)$ and stores (t', f') on any two servers that are now available, not necessarily the same ones it read the file from. More precisely, it writes to all three servers, and when it hears back from the first two servers, then it outputs $(\mathsf{changed})$.

We can now prove that the system is consistent. The trick is to prove the following invariant:

- After n change commands there exist a file f such that (n, f) is stored on at least two servers;
- The third server stores a value $(n', \cdot)$ such that $n' \leq n$.

First assume the invariant is always true. Then it is easy to prove consistency. Namely, when the client reads from two servers it will get (n, f) and (n, f) or it will get (n, f) and (n', f'). In both cases it will output f since $n' \leq n$. What is left now is to prove the invariant: Clearly the invariant holds after 0 change commands as all three servers store $(0, \epsilon)$. Assume then that the invariant is true after $n-1$ change commands. This means that at least two servers are storing $(n-1, C^{n-1}(\epsilon))$ and the last server stores (n', f') with $n' \leq n-1$. So the client will write back $((n-1)+1, C_n(C^{n-1}(\epsilon))) = (n, C^n(\epsilon))$ on two servers, maintaining the invariant.

△

Exercise 1.1 (Generalising Replication to More Parties) We generalise the protocol in Example 1.2 to n servers.

1. Let r and w be positive integers. To read the file, read (t, f) from r servers and use the file with highest timestamp. To change a file, read from r servers and write back to w servers. Argue that this works if $r + w > n$. We call r the read quorum and we call w the write quorum.
2. We can generalize further. Assume the servers are named $0, \ldots, n-1$ and let $[n] = \{0, \ldots, n\}$. Let $2^{[n]}$ be the power set of $[n]$, i.e., the set of all subsets of $[n]$. Let $\mathcal{R} \subseteq 2^{[n]}$ be the read access structure. Let $\mathcal{W} \subseteq 2^{[n]}$ be the write access structure. To read, read (t, f) from all servers in any $R \in \mathcal{R}$. I.e., read (t, f) from some servers and let R be the set of servers you heard back from so far (say the fastest to respond). Continue to read until $R \in \mathcal{R}$. Then use the file with highest timestamp. To change, read from all servers in some $R \in \mathcal{R}$ and then write to all servers in some $W \in \mathcal{W}$. What is the minimal condition on the pair $(\mathcal{R}, \mathcal{W})$ that will make the system consistent?

△

1.3.4 Types of Failure

The main subject of a course on secure distributed systems is to construct systems which remain secure even in the presence of errors or failures. Constructing a system which works when the network is fast and delivers all messages, when all machines are present and everything runs correctly, and when no attacker is present and bandwidth is unlimited is fairly easy. But that is not how the world looks. Messages are lost, message delivery is slow, messages are tampered by adversaries; bandwidth is limited, machines are slow, physical clocks drift, machines crash, machines are taken over by hackers or subjected to denial of service attacks. Such deviations from the expected or normal behaviour are sometimes called failures. Making a distributed system which behaves correctly even in the presence of such failures is sometimes called *fault tolerance*.

There are several reason why fault tolerance is mainly a subject of interest in distributed systems. First of all, as you add more machines, the probability that any one of them fails will go up, so you simply cannot afford to build a system which fails if any of the components fails. Also, if you are running on a single machine and the machine crashes there is little you can do. But distributed systems can have partial failures; for instance in a system with three machines when a single one crashes, it is no longer obvious that you cannot keep running. So you should try to make a system that does exactly that. Otherwise your competitors will do that and you will be out of business! Lastly, since the machines coordinate over a network, your will rely on even more machines (IP routers, DNS servers, ...) and will be subject to attacks by adversaries on that network. You cannot ignore these possible network failures and attacks. Today these attacks are ubiquitous and persistent. If your system does anything useful, eventually someone will try to break it. And if you designed your system in an insecure way, the adversary will succeed.

Note that it is not always clear what constitutes a failure and what is considered as normal behaviour. The fact that a UDP package is dropped by the network is not really a failure, as it is within the specified behaviour of the UDP layer. You are expected to handle that yourself or use TCP/IP instead. The fact that a TCP/IP connection can be dropped is not really a failure either, maybe the machine in the other end was under such computational load that it could not handle the connection, and these things are expected to happen as load goes up and down. However, it still makes sense to divide the behaviour of the environment of the system into normal conditions and rare conditions. One then tries to design the system such that it works well and fast under fair-weather conditions. One then in addition ensures to handle the rare cases in a graceful manner, for instance by just getting slower or shutting down. Essentially the rare cases are not allowed to result in safety errors like leaking data.

That means to specify what your system is supposed to do, you define the system's *specification*, or *input-output behaviour*, for each set of conditions that the network and machines involved may find themselves in. This defines the behaviour of the system under the given conditions. For instance: "When all machines are present and their clocks synchronised within 1 second and all messages are delivered within 1 second, then the system delivers a version of the stored file which is at most three second old. If one machine crashes and ..., then it takes 10 seconds to deliver. Under all other conditions no guarantees are provided."

That said,we will fix the terminology for some events that are considered rare. We will follow standard terminology and refer to them as failures.

Crash failure: A machine or process crashes, it stops running now and will never come alive again. There are three main models for this.

- **Fail-arbitrary:** As the process crashes, the last message it sends is some arbitrary possibly faulty message.

- **Fail-silent:** As the process crashes it sends out no message to any other process.
- **Fail-stop:** As the process crashes it sends Stop to all other processes. This is an example of a theoretical failure type. It doesn't happen in practice, but as we discuss below, you can sometimes reduce real-life failure to these theoretical ones.

Crash-recovery failure: A machine or process crashes, and stops running. Later it comes alive again. Maybe someone accidentally turned it off and later turns it on again. Maybe it burned and is later replaced by a fresh machine by the local system administrator. The crash part of crash-recovery can come in the same three flavours as above defined for crash failures. When the machine comes back to life there is typically some re-entry protocol that allows it to catch up on what happened. Crash-recovery failures are harder than crash failures, as it is typically expected that when the machine comes alive again, it should start contributing to the system again, taking inputs and giving outputs. When a machine crashes and then recovers, it is assumed that all values in volatile memory (RAM) are lost. It is typically assumed that values on permanent memory (disk) are uncorrupted.

Reset failure: A machine or process is reset to some previous state. It, for instance, could forget that it already sent a given message, and so may send it again. This is another example of a theoretical failure type.

Timing failure: A machine does not run as fast as expected and so sends a message too late – or runs too fast and sends it too early.

Message delay failure: A sent message does not arrive within the time expected; the network is slow.

Byzantine process failure: A machine or process is replaced by another process running a possibly maliciously chosen algorithm. This can be due to programming error, hardware error or because the machine is taken over by a hacker. There are two main flavours:

- **Monolithic:** If several machines are Byzantine corrupted then we assume that there is a central entity (often called the adversary) who controls and coordinates all of them. Think of this as a hacker who broke into all of them and has a network connection to all of them allowing him to orchestrate how they misbehave. We assume that the adversaries ability to coordinate is unlimited, so he is not restricted to using the available network resources. It can magically see the state of all corrupted machines and instantaneously instruct them to send nasty messages or do nasty stuff.
- **Local:** Each Byzantine party had its code replaced by some malicious code, but they have no magical means for colluding, they have to do that via the network available to the honest correct machines.

Duplication failure: A message that should have been sent only once is sent multiple times.

Injection error: A message that should never have been sent was sent, perhaps sent over the network by an adversary trying to break your system.

Tampering failure: A message was changed while being sent, possibly by an adversary trying to break your system.

Omission failure: A message that should have been sent was never sent, or it was dropped on the network, or not received at the receiving end.

Example 1.3 (Reducing Between Error Types) As we mention above, there are tricks for reducing one type of error to some other type of error which is easier to handle. For instance, crash-recovery failure can be reduced to reset failure if you have permanent storage: now and then you write your current state to disk. If you crash and recover, you then retrieve the latest state from disk and start running from that state again. Reset errors can then be further reduced to duplication errors: you could make sure to write your full state to disk every time you are about to send some message m. After sending m, you write $(\mathsf{I\ sent}, m)$ to disk. If you crash and recover, read the latest state from disk. If running from this state would mean you send m, check the disk again. If there is no message $(\mathsf{I\ sent}, m)$ then send m again and write $(\mathsf{I\ sent}, m)$ to disk. Otherwise, run from the retrieved state but ignoring the instruction to send m. If you are extremely unlucky, it could happen several times that you send m but crash before you can write $(\mathsf{I\ sent}, m)$ to disk (maybe it is the attempt to write to disk which crashed the process). But this is the worst that can happen. And so crash-recovery errors have been turned into duplication errors. As a last example, crash-silent errors can be reduced to crash-stop errors if you have precise clocks and a bound on message delivery time: each second each machine sends to each other machine a Ping message. When a machine stops sending Ping, then interpret it as the machine having sent Stop.

Exercise 1.2 (Crash recovery) In this exercise you are asked to add crash-recovery resilience to the system from Example 1.2. Assume that there is a reliable way to detect when one of the three servers crashes. When this happens the local system administrator will restart it or replace it with a new server. In both cases it will be restarted with the state $(-1, \epsilon)$ so that it knows it has been crashed. Then the server will do some protocol where it talks to the other servers, called the *reentry* protocol. By definition this protocol ends when the server outputs reentry. During down-time the server does not respond to requests from the client. At the end of the reentry protocol the server is considered to be back in the group of servers and will again respond to the client as in Example 1.2. Describe a reentry protocol that will make the system consistent under the following conditions. Denote by the down-time interval of a server the time interval in which it is crashed plus the time interval it takes to reboot or be replaced plus the time interval it takes to run the reentry protocol. Your system should be secure as long as no two servers have overlapping down-time intervals. Notice that there is a trivial solution which makes the reentry protocol simply be an infinite loop. Try to do better.

△

1.3.5 Correctness, Consistency, Fault tolerance, Security: A rose by any other name would smell as sweet

Different subfields of distributed systems use a number of different terms for the simple notion that a system does what it is supposed to do under some given failure model, and nothing more. And sometimes the failure model is called the attack model or the adversary model. To reduce confusion, we will fix a terminology. We will use attack model to denote the specification of the things that can happen to the system. So the attack model could for instance be "fail-stop crash failures". To specify what a system is supposed to do, we need to specify for at least one attack model what the system is supposed to do. For instance, under the crash-recovery model where at most one out of three servers is down at a time, the system will deliver consistent files. In general we say that under some attack model $\mathcal{A}$ the system behaves according to specification $\mathcal{S}$. Then $(\mathcal{A}, \mathcal{S})$ is the full specification of the system's guarantees. This is to some extend just a rephrasing of the notion of a threat model and security policy. But there are some qualitative differences. A security model tends to focus on the things that should be avoided, listing the bad things that are not allowed to happen: How the system is actually meant to operate in normal conditions is phrased in functional specifications. The above specification view instead specifies what the system *must* do even when under attack. Everything else is implicitly left as faulty behaviour. Security models also tend to focus on the physical system itself: we don't want viruses and unauthorised access to the machines. The above specification view is used when specifying security not for a physical system but for a protocol like a middleware layer. In that sense they complement each other.

In the above specification model we say that a protocol π is a secure implementation of its specification if whenever it is run, even when under an attack allowed by $\mathcal{A}$ it behaves according to $\mathcal{S}$.

This means that security is always relative to some specification: It does not make sense to say that a system is secure without saying what it is supposed to do and under which circumstances. Notice in particular that if a secure protocol is run in an attack model that is not allowed by $\mathcal{A}$, then all bets are off, even if π is a secure implementation of $(\mathcal{A}, \mathcal{S})$. The notions of correctness, consistency and fault tolerance are all just notions of security according to some specification. Correctness is typically just achieving security in an attack model with no or very mild failures. Consistency is simply security in the context of databases and replicated systems. Fault tolerance is security in attack models with bad failures. To avoid confusion, we will stick to this notion of security throughout the text and always be very explicit about what the attack model and specification are.

1.3.6 Specifying and Modelling Distributed Systems

We now return to exactly how we specify what a system is supposed to do and what an implementation is actually doing.

Formally, our systems will be specified via interactive agents (IAs). An interactive agent A is a computational device which receives and sends messages on named ports and which holds an internal state.

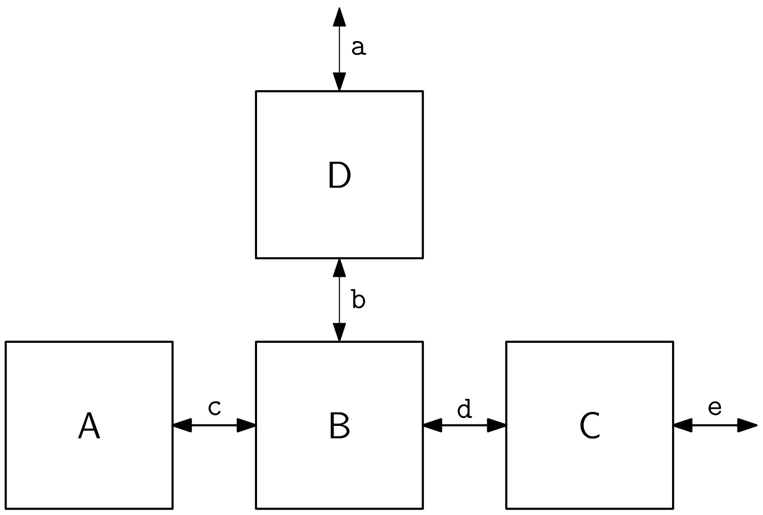


Figure 1.6 An interactive agent A with a port c; An IA B with ports b, c and d; An IA C with ports d and e; And an IA D with ports b and a. Together they make up an interactive system with open ports a and e

When an IA is run it will produce a trace, which is just the sequence of messages exchanged between the agents in the order they were exchanged. Formally: start with the empty trace, and when m is written on port p you append (SEND, p, m) to the trace. When m is read on port p you append $(\text{RECEIVED}, p, m)$ to the trace. Most of the specifications we will consider in this book can be defined via traces. This is done via trace properties, which are just functions P which take traces as input and return true or false. If τ is a trace, then $P(\tau) = \top$ means that τ has the property P and $P(\tau) = \bot$ means τ does not have the property P. There are two important classes of trace properties:

Safety property: A safety property is something which is true until some safety condition is broken, at which point it never becomes true again. More formally, a safety property is a trace property P such that $P(\epsilon) = \top$, and if $P(\tau) = \bot$, then $P(\tau \circ \tau') = \bot$ for all traces τ'. Here ϵ is the empty list of inputs and outputs. An example of a safety property could be "no machines in the systems has output 42". This is clearly true before the system has begun running, and when the property was broken it cannot be undone.

Liveness property: Liveness properties are something which will eventually become true (in some attack model). A bit more formally, a liveness property is a trace property P such that $P(\epsilon) = \perp$, and if $P(\tau) = \top$, then $P(\tau \circ \tau') = \top$ for all traces τ'. An example of a liveness property could be “some machine has output 42”. This is clearly false before the system has begun running, and when it happened once it cannot be undone.

It is natural to think that the formalisation of liveness given above is wrong. Assume for instance that we want to say that some machine outputs 42. Should it not instead be something along the line of $P(\epsilon) = \perp$ and $\exists \tau < \infty$ such that $P(\tau) = \top$? Then the property would more intuitively say that some machine will eventually output 42. The reason why this does not work is that at a point where the system only ran for some finite time, it might still be that no machine output 42. So the trace does not have the property that $\exists \tau < \infty$ such that $P(\tau) = \top$. Yet, if we would let the system run a bit longer, then maybe the trace would at some point get the property. When we want to say that a system has a given liveness property, we therefore need a bit more care. We basically need to say that for all runs of the system where there is no more meaningful work to be done (like messages that need to be delivered) and where all the network resources that the protocol used also were live, the system make the liveness predicate true. We will in Section 3.8 discuss in more detail the subtleties of specifying liveness properties. The above definition simply specifies the minimal requirement, that a liveness property is something that once a good thing happened, it does not go away again, like the arrival of an expected message.

It is common in distributed systems to distinguish sharply between safety and liveness properties. You should be careful about your safety properties, if you break them it can never be undone. For liveness properties you can be more lax, as if you do not manage to fulfil them now, there might always be a chance later. It turns out that any trace property can be written as a conjunction of safety and liveness properties. More importantly, this can often be done in a natural way.

Consider for instance this property: “Eventually exactly one machine will output 42.” It is not a liveness property, as it can become true, but then false again (when the second machine outputs 42). It is not a safety property either as it is not true from the beginning of time, where no machines have output anything yet. However, it is a conjunction of these trace properties: “Some machine has output 42” and “It did not happen that more than one machine has output 42”. The first one is a liveness property. The second one is a safety property. When they are both true, then exactly one machine has output 42.

Exercise 1.3 (trace properties I) For each of the following trace properties, determine whether they are liveness properties, safety properties or neither.

1. If S gets input $(\textsc{Send}, m, R)$, where R is the name of a receiver, then R will at some point output $(\textsc{Received}, m, S)$.

2. If R outputs $(\textsc{Received}, m, S)$, then at some earlier time S got the input $(\textsc{Send}, m, R)$.
3. If R outputs $(\textsc{Received}, m, S)$, then no other party will ever output $(\textsc{Received}, m, S)$.
4. If R outputs $(\textsc{Received}, m, S)$, then all other parties will eventually output $(\textsc{Received}, m, S)$, and no other parties will output anything else but $(\textsc{Received}, m, S)$.

△

Exercise 1.4 (trace properties II) For each of the following trace properties, write them as a conjunction of liveness properties and safety properties.

1. R outputs $(\textsc{Received}, m, S)$ if and only if at some earlier time S got the input $(\textsc{Send}, m, R)$.
2. If R outputs $(\textsc{Received}, m, S)$, then all other parties already output $(\textsc{Received}, m, S)$, or will do that the next time they send something

△

1.3.7 Types of Distributed Systems Models

There are many ways to model the network and machines of a distributed system. The main distinction is on whether the protocols use a notion of physical time and how.

Fully synchronous: In a fully synchronous system, each process P has a clock $c(P)$ and there is a known bound Δ_{CLOCK} on how far they drift from real time. There is also a known upper bound Δ_{SEND} on how long it takes to send a message between any two parties. For sake of definition, the model often assumes some notion of global time, which is used to define what the real time is. Of course there is no such thing in real life due to Relativity theory, but for distributed systems located on the surface of the earth UTC is a pretty good approximation of some global time. We use N to denote nature and use $c(N)$ to denote the real time: the time of nature. So, if we define $c(X)$ to be the clock at process X, then it holds for all processes P that $|c(P) - c(N)| \leq \Delta_{\text{CLOCK}}$. Furthermore, if a message is sent at time $\text{snd}(m)$ on the clock $c(N)$ and received at $\text{rcv}(m)$ on the clock $c(N)$, then $\text{rcv}(m) - \text{snd}(m) \leq \Delta_{\text{SEND}}$. We look in much more depth at physical time and synchronous protocols in Chapter 8.

Fully asynchronous: In a fully asynchronous system, the processes P do not even have clocks, so the protocols cannot be specified by asking parties to do things at specific times. Also, there is no known upper bound on how long it takes to send a message. In fact, for each message the adversary might decide to completely drop it, or deliver it at any time of her choosing. An example of an asynchronous system is the HTTP protocol: When a client wants a web page it sends a GET request to the server and opens a port where it expects the data back. Then it just waits until the data comes. The server, when it gets the GET request retrieves the page from disk (or constructs it) and then sends it back when it is ready. We look in much more depth at asynchronous protocols in Chapter 9.

Partially Synchronous: In the partially synchronous model[1] it is assumed that if a message is sent by a correct process and that process never crashes, then the message is *eventually* delivered. There are several ways to model this, and often people do not even take the care to make precise what they mean by "eventually". Here is one common way to do it: There is a number Δ_{SEND}, some length of time. If a message is sent by a correct process that does not crash, then it is delivered within that bound. But the value Δ_{SEND} can be arbitrarily large and it is *not known to the distributed system.* (One way of thinking of this is that you first design the system, and then some evil entity sets Δ_{SEND}). So what the synchronous model and the partially synchronous model have in common is that messages always get delivered within some time bound. The difference is that in the synchronous model the protocol designer is assumed to know the upper bound on network delivery. In the partially synchronous model the protocol designer is not allowed to use the bound, it is merely assumed to exists. It seems odd that knowing that a secret upper bound exists should be useful, but when we cover Finality Layers for Blockchains in Chapter 12, we will see an example that one can do more in the partially synchronous model than in the fully asynchronous model.

Eventually synchronous: Another popular mixed model is the eventually synchronous model, where the processors have clocks. Also two bounds Δ_{CLOCK} and Δ_{SEND} are given. It is assumed that the network in some periods is asynchronous: the clocks can be far apart and messages take arbitrarily long time to be delivered. However, it is assumed that eventually there will always come a period where the network is synchronous, i.e., during this period it holds for all processes P that $|c(P) - c(N)| \le \Delta_{\text{CLOCK}}$ and all messages m sent and received in this period that $\text{rcv}(m) - \text{snd}(m) \le \Delta_{\text{SEND}}$. The protocol does not know when the period has arrived. So what the synchronous model and the eventually synchronous model have in common is that there is a known upper bound on message delivery. The difference is that in the synchronous model the messages are always delivered within this bound. In the eventually synchronous model they are eventually delivered within this bound for a long enough period. This allows us to set the bound more liberally in the eventually synchronous model.

It can be fairly easy to design systems for the fully synchronous model. However, the systems are very hard to implement in practice as you have to constantly synchronize the clocks of the involved parties and find a way to make sure all messages that are sent are delivered within the bound. This can be hard during periods where machines crash or the Internet is slow. Moreover, all machines engage in several systems with several different and overlapping groups of servers. So if all these protocols had to be synchronous, in the end the entire Internet

[1] This model was introduced in [4].

would have to be synchronised, which is completely infeasible as machines are constantly crashing, messages are being dropped and denial-of-service attacks are being mounted. Therefore the synchronous model is mostly of academic interest: it is the easiest test-bed for designing protocols. It can also be relevant for systems running in a very closed environment, like three database servers within the same administrative domain. However, it is one of the hard won insights in distributed systems that when you describe your system you should never refer to any notion of real time. Protocols relying on real time by experience tend to break or be very slow in practice.

In comes the asynchronous model. This model does not even give you a clock to refer to and there is no bound on message delivery time. So if you can design a system that is secure in the asynchronous model, it will also be secure when run in practice on any network. The flip side is that it is very hard to design systems for the asynchronous model. When you have no notion of time you can for instance not distinguish between a server that is crashed and a server which just has messages that take arbitrarily time long to reach you. So if your system has some special server that is important for liveness, you cannot know when it is down and so cannot know when to replace it. This sometimes makes it impossible to solve in the fully asynchronous model tasks which can be solved in the fully synchronous model.

Hence the eventually synchronous model. The intended use of this model is as follows. Assume you have a system which is specified by some safety properties $\mathcal{P} = \{S_1, \ldots, S_n\}$ and some liveness properties $\mathcal{L} = \{L_1, \ldots, L_m\}$. You are supposed to design your system as follows. The safety properties $\mathcal{P}$ will never be broken, neither in synchronous nor in asynchronous periods. The liveness properties need not become true in asynchronous periods, but if the system become synchronous for long enough, then your system has to be live, i.e., you for instance start to deliver messages to all receivers. For such a system, clock synchronisation and the bounds Δ_{CLOCK} and Δ_{SEND} can be set to be fairly small, and clocks can be synchronized only rarely. If the clocks drift too far in some period or messages arrive very slowly, it is rarely a big issue as safety will not be broken, and we can assume that clocks will probably soon be synchronized again, and the network will be back to fair-weather conditions and so will start delivering again.

Similarly the partially synchronous model is a good balance between the to theoretical synchronous model and too demanding asynchronous model. As mentioned above, we will return to this model when we study Finality Layers.

Exercise 1.5 (detecting fail-silent crashes) Assume that process P is supposed to send a message m to Q at clock time t, i.e., when $c(P) = t$. Assume that there are no omission errors but that P might fail-silent crash before sending the message. What is the earliest time (measured using $c(Q)$) at which Q can conclude that P must have crashed?

1.3.8 Thinking like a Cryptographer in Distributed Systems

How should you pick your attack model? Well, the first answer is that you should take a look at your threat model. Your threat model specifies which attacks you find possible, and that should be your attack model when you analyse your system. The second answer is that you should be conservative. If you are not completely sure that your machines will never crash, you should add process failure to the attack model. If you are not completely sure that when the machine crashes it might not make the operating system send some weird message, incorporate fail-arbitrary. Or even better, maybe you are not completely sure one of your machines cannot be taken over by a crazy hacker. In that case you must let the attack model be Byzantine process failures. Sometimes you would like to be liberal when you pick the model as it will make your life easier. Do not do that. Machines will crash, messages will be seen by the adversary, and some machines will eventually be broken into. Deal with that already when you design your system.

1.3.9 Canonical Goals

So now we know how to design distributed systems and how to specify what we want them to do under which conditions. But what should we make our system do? It turns out that no matter which system you try to build, there is a small collection of problems that keep popping up. Even better, if one has a solution to these problems, then building most other systems on top is pretty easy. These are the power tools of distributed systems. We will study the most important ones. They also happen to be didactically complete. If you understand how to implement these, you have learned to think in a distributed way.

Leader Election: In many distributed systems, one can gain efficiency by having one server take on a special role, like collecting and distributing tasks. Such a server is often called the leader. The problem is that when the leader crashes, liveness typically breaks down in a very bad way. Then it is time to pick a new leader. This is called leader election. The goal is that all processes end up agreeing on some new non-crashed process that will be the new leader. We look at leader election in Chapter 12.

Broadcast: You want to send a message m to all machines in the system. This is easy when there are no failures. If there are transmission errors, various systems with acknowledgements and resends will solve the problem. In case of process failures, the problem becomes much harder. It is typically needed that all processes receive the same message, even if the sender and some of the other processes crash or are suffering Byzantine failure. The problem can be even harder when considering many messages being sent by different parties at the same time and require that they arrive in the same order at all processes, known as totally ordered broadcast. We look at Totally Ordered Broadcast in Chapter 11.

Consensus: All processes have a local input bit 0 or 1. This could for instance be a bit indicating whether they are all ready to proceed in the protocol. They should end up outputting a common decision, i.e., there is a bit b such that all correct processes eventually output b. There are several variations on how this output depends on the inputs. A popular one to study is this one: If all correct processes vote 1, then the output should be 1. If all correct processes vote 0, the output should be 0. Otherwise the output can be arbitrary, as long as all correct processes agree on the output. We look at Consensus in Chapter 8 and Chapter 9.

MUTEX: Mutual exclusion. There are several servers in a system. The processes run the same code. There is some critical region in the code that only one server is allowed to enter at a time. This problem is already hard in multi-threaded programs. It becomes much worse in a distributed system where the processes are physically separated.

Here are some problems that can be solved with these tools.

Work distribution: You have a system where requests come in much faster than they can be solved by one machine. You make ten copies of the system and set up a special entry server, which only receives the requests and then distributes them to the workers in a balanced manner. The workload balancer crashes. The workers notice that work stop coming in. They run a leader election and the winner is promoted to the new entry server.

State machine replication: You have a critical program M that is so important that it must keep running even if the machine it runs on crashes. You then simply run it on n machines. When clients want to give an input, it is sent to all copies of m. If you use totally ordered broadcast, secure under Byzantine process errors, to broadcast the inputs to all replicas, then they will run M on the same inputs in the same order and therefore keep a consistent copy of M. Then they broadcast the outputs of M back on the totally ordered broadcast channel. The Byzantine machines might send wrong outputs on the broadcast channel, but as long as less than $n/2$ of the servers are corrupted, the correct output can be found just by taking majority of the proposed values. So this runs a consistent copy of M that can tolerate less than $n/2$ Byzantine failures.

Recovery: There has been a very bad event in your state machine replication system where a lot of machines crashed and the network completely broke down. Now all machines are back alive and the network healthy again. You wonder whether all the machines hold the same state for M. Let M_i be the state held by party i. All processes send their state M_i to all other processes. Process j lets $b_j = 1$ if and only if $M_1 = \ldots = M_n$. They might get different results if some of the machines are still Byzantine corrupted and report different M_i to different parties. Therefore they run Consensus on $(b_1, \ldots, b_n)$ to get a common decision b. If $b = 1$ they continue running. If $b = 0$, then they must start some clean up procedure. Note that if the output is $b = 1$, then at least one honest process had input $b_i = 1$ and therefore all correct processes hold the same state for M. So it is safe to keep running. Also, if all processes are correct and hold the same state $M_i = M$, then the output will be 1, so they keep running from M.

Sharing exclusive physical resources: A large number of machines share a common physical broadcast channel, like a cable or some frequency for Wi-fi. If two machines send at the same time, both signals are lost. Run a distributed MUTEX protocol.

2

Communication

Contents

2.1 Introduction

The glue of a distributed system is communication, which is the subject of this chapter. We will see how the Internet allows you to communicate between two entities and how to organize communication in systems with more than two entities, including a toy example of a peer-to-peer network. We will also touch upon naming: how you give names to entities in a distributed system and how you locate them given that name. This is what the DNS system does on the Internet. We will also take a look at consistency of communication: when several parties send messages to each other concurrently, how do we ensure that they have a consistent view of what goes on? We will in particular look at how to use so-called vector clocks to implement causal communication. In causal communication we want that if a message m_2 might depend on some other message m_1, then the communication layer delivers m_1 to the application layer before m_2 at all correct processes.

There are two main types of communication:

Message Passing: Processes send messages to each other and the transfer is typically between two only parties, the sender and the receiver, and the sending and the receiving are separated in time. If you communicate by playing a drum or sending smoke signals, it is message passing. E-mail and TCP/IP are also examples of message passing.

Shared Memory Space: Processes are in a setting where they can see the same variables. They communicate by writing and reading these variables. Different cores and threads on the same computer communicate via a shared memory space (the memory of the computer). Applications built on top of a database work via shared memory space, the database. In the introduction we hinted at how to implement a shared variable using message passing, by using appropriate quorums for reading and writing.

Both types of communication are worthy of being the subject of an entire book on their own. However, the course for which this book is written does not allow us the time to look into both of them in depth. Since many of the underlying principles of achieving secure distributed systems can be learned by studying either type of communication, and we want to go into some technical depth, we have chosen to focus on one of them. We have chosen message passing as we believe it will be an easier introduction to the central concepts. In addition, it allows us to set the scene for a chapter on blockchains and cryptocurrencies, which we believe any modern book on secure distributed system should cover.

Another common distinction between types of communication is as follows:

Transient: The receiver has to be alive when the message is sent to see it. If the receiver is not alive when the message is sent, the message will be lost. An example is TCP/IP.

Persistent: The message can be seen even if the receiver comes alive later. Examples are programs running on top of databases and e-mail.

Another common distinction between types of communication is as follows:

Synchronous: When sending a message, the sender waits until the receiver received the message before it continues its own execution. Similarly, if the receiver is to receive several messages, it receives them sequentially. This is a simple way to program as there is not so much concurrency, but it can be terribly slow. If it takes a second to send and process an element, and you have to send 100 elements from the same sender to the same receiver, it can be minutes instead of seconds if you do it synchronously.

Asynchronous: When sending a message, the sender continues its own execution while the message is being sent. Similarly, if the receiver is to receive several messages, it does so using seperat threads. This can speed up applications considerably. For instance, when you download a page using HTTP, then each element, for instance pictures, are sent in parallel from the server to the browser. This can speed up the application considerably. The flip side is that it is harder to program in this style.

A final common distinction between types of communication is as follows:

Push: Signals propagate by the senders actively sending them. This is a good communication pattern if messages come at random times or receivers are few and known. It is also a good pattern if you are sure the receiver needs the data. A canonical example is e-mail.

Pull: Signals propagate by the receivers asking for them. This is a good pattern if message come at regular times or readers are many or unknown. It is also a good pattern if not all clients might need the data. An example is reading your favourite news site in the morning.

These patterns can be combined. You might for instance push a small notification to all clients that a large update is available. Then the clients will pull the update if they need it.

Exercise 2.1 (Shared Variable Space from Message Passing) Revisit Example 1.2 on replication. Recall that we considered only one client there. If there are two clients, the system does not work. It could happen that client 1 reads the current fill, applies change $f_1 = C_1(f)$. Then client 2 reads the current file (before client 1 wrote back its change), applies change $f_2 = C_2(f)$. Then client 1 writes back f_1 and gets confirmations. Then client 2 writes back f_2 and gets confirmations. Now client 1 considers the change as successful, yet C_1 was never applied. The value stored in the system is $C_2(f)$. What we want if two clients each make a change is that the end state of the system stored either $C_1(C_2(f))$ or $C_2(C_1(f))$. Try to design a system which tolerates this if all servers and clients are correct. Can you make it work even if there could be one Byzantine error among the servers? Can you make it work if a client might be Byzantine corrupted?

2.2 Let's Go!

We are going to present code examples in the Go programming language as it is well suited for distributed programming and allows for concise code. It is a static typed language. It is object oriented without inheritance. All types can be used

```go
package main

import ( "fmt" )

// A function declaration: Types after variable names; Multiple return values
func multi(x int, y int) (string, int) {
     return "The answer is", x+y
}

// A class
type Named struct {
     name string // Member of a class
}

// Method on a class
func (nm *Named) PrintName(n int) {
    if n < 0 { panic(-1) } // an "exception"
    for i:=0; i<n; i++ {
        fmt.Print(nm.name + "\n")
    }
}


func main() {
    var i int // Explicit declaration of variable
    i = 21    // = is used for assignment
    j := 21   // := declares the variable based on the type of the value

    decr := func() int { // a closure
        j = j-7
        return j
    }

    str, m := multi(i, j)
    defer fmt.Println(str, m) // run after return or a panic

    fmt.Println(decr())
    fmt.Println(decr())

    nm1 := Named{ name: "Jesper" } // value
    nm2 := &Named{} // pointer

    nm1.PrintName(2) // Calling a method
    nm2.PrintName(2) // . does auto-dereference of pointer
    nm2.name = "Claudio" ; nm2.PrintName(2) // RETURN or ; as seperator
    var nm3 *Named = nm2; // how to explicitly declare a pointer
    nm2.name = "Ivan"
    nm3.PrintName(2)
    nm3.PrintName(-1) // let us cause a panic
    fmt.Println("Will we make it here?")
}
```

Figure 2.1 Go by example.

as values or pointers. It allows for first order functions, a.k.a. lambda closures. If you programmed in C, C++ or Java, the syntax should not surprise you, except that it takes a while to get used to the type being mentioned *after* the name. In Fig. 2.1 there is an example of a Go program.[1] When it is run it produces the following:

```
14
7
Jesper
Jesper


Claudio
Claudio
Ivan
Ivan
The answer is 42
panic: -1
<program terminated>
```

2.3 The Internet

In this section we will briefly take a look at how we come from a situation where some machines are connected by copper cables and electromagnetic waves to a situation where any machine on the Internet in principle can send a messages to any other machine on the Internet. The explanation will be very short indeed. This is not to underestimate the importance of this engineering achievement: Creating the Internet is one of the biggest engineering achievements of humanity. However we only scratch the surface this important topic. First of all, we assume the readers of this book have already taken courses covering the basics of how to build and use TPC/IP and DNS, the backbone(s) of the Internet. Second once the Internet gives you the ability to send a message between two machines, thus establishing the smallest meaningful distributed system—two machines—and the simplest distributed task—sending a message, another equally daunting task remains ahead of us: to use this ability in a principled way to construct secure distributed systems consisting of a large collection of machines acting in concert to solve some distributed task like implementing a blockchain. We believe that studying this problem most effectively teaches the principles which will allow you to solve the problems you will meet in designing, programming, and using the distributed systems of today and tomorrow.

We expect that the reader has familiarity with the OSI (Open System Interconnectivity) model, but will recap the main layers in this section for self containment.

[1] For a Tour of Go, see for instance `https://tour.golang.org`.

```
package main

import ( "net" ; "os" ; "fmt" ; "strconv" )

func main() {
     name, _ := os.Hostname()
     addrs, _ := net.LookupHost(name)
     fmt.Println("Name: " + name)
     for indx, addr := range addrs {
              fmt.Println("Address number " + strconv.Itoa(indx) + ": " + addr)
     }
}
```

Figure 2.2 A Go program which will display the name and IP address of your machine. Try the program to make sure you can run Go. Also try to understand the output of the program. We take the chance to introduce Go a bit more: When we use an underscore as a receiving variable it just means that we will not be using that value. The `range` keyword can be used to iterate over elements in a variety of data structures. Here `addrs` is an array of strings (containing the possible names of the host). In the loop `indx` iterates over all entries in the array and `addr = addrs[indx]`.

2.3.1 Physical Layer

There are several physical media for sending a message from two geographically separated machines, like WIFI, copper cable and pigeons. In the ISO/OSI model this is called the physical layer. It is used to move a raw unstructured package of data between two physically connected locations. How this is done depends on the underlying technology.

2.3.2 Data Link Layer

The data link layer provides a mean to send a stream of data from one node to another, including how to establish, maintain and close a connection. It detects (and possibly corrects) errors that may occur in the physical layer. But at this layer connections can only be established between two nodes that are physically connected via the physical layer. It is in turn broken into two layers:

Medium access control: Tshe MAC layer is responsible for controlling how devices in a network gain access to a medium and permission to transmit data.

Logical link control: The LLC layer is responsible for identifying and encapsulating network layer protocols, and possibly has mechanisms for error detection and retransmission.

```go
package main

import ( "fmt" ; "net" )

func main() {
    ServerAddr, _ := net.ResolveUDPAddr("udp",":10001")
    ServerConn, _ := net.ListenUDP("udp", ServerAddr)
    defer ServerConn.Close()
    buf := make([]byte, 1024)
    for {
        n,addr,err := ServerConn.ReadFromUDP(buf)
        fmt.Println("Received ",string(buf[0:n]), " from ",addr)
        if err != nil {
            fmt.Println("Error: ",err)
        }
    }
}
```

Figure 2.3 Here a UDP server receives UDP packages and prints them and a UDP client sends some packages to be printed. It listens on port 10001, which is just an arbitrary port we picked for this example. Try running them on your machine to make sure Internet works on your machine. Try running several clients at the same time. Try modifying the client and server to make them do something more interesting to get used to Go and IP. We learn a bit more Go: In Go arrays are pointer types, so when you declare an array, it is `nil` and if you try to index into it you will get a panic. Therefore,c first you have to make an array of the right type and store the variables in it before you can use it. This is what happens in `buf := make([]byte, 1024)`.

```go
package main

import ( "net" ; "time" ; "strconv" )

func main() {
    ServerAddr, _ := net.ResolveUDPAddr("udp","127.0.0.1:10001")
    LocalAddr, _ := net.ResolveUDPAddr("udp", "127.0.0.1:0")
    conn, _ := net.DialUDP("udp", LocalAddr, ServerAddr)
    defer conn.Close()
    i := 0
    for {
        i++
        msg := strconv.Itoa(i)
        conn.Write([]byte(msg))
        time.Sleep(time.Second * 1)
    }
}
```

Figure 2.4 A UDP client sending packages to port 10001 on the local machine. If the server is running on another machine, the `127.0.0.1` would have to be replaced by the IP address of that machine.

```
package main

import ( "net" ; "fmt" )

func main() {
  ln, _ := net.Listen("tcp", ":")
  defer ln.Close()
  _, port, _ := net.SplitHostPort(ln.Addr().String())
  fmt.Println("Listening on port " + port)
}
```

Figure 2.5 A Go program which starts listening on a random local port.

2.3.3 Network Layer

As the name indicates, the network layer provides the functional and procedural means for sending data from one node to another node to which it is connected only indirectly via a network of other nodes across "different networks". The network layer sends so called packets i.e., variable length data sequences. To be able to send packets to a node somewhere else in the network, every node has an address. When you want to send a packet to some address, you simply give the address and packet to your OS which will then use the network layer to find a way to deliver the message to the destination node, possibly routing it through intermediate nodes. If the message is too large to be transmitted from one node to another on the data link layer between those nodes it will be split into several fragments at one node, each fragment is sent independently, and all of them are reassembled at another node.

The best known Network Layer is the Internet Protocol (IP) protocol. In IP (version 4) an address consists of 32 bits, typically shown as 4 octets of numbers from 0–255 represented in decimal form, for example `198.41.0.4`. The address `127.0.0.1` has the special semantics of always pointing to the local machine.

We assume that the reader has a rudimentary understanding of IP and UDP, for example from a course on networks and operating systems.

2.3.4 Transport Layer

In OSI the transport layer is meant to transfer data sequences of varied lengths from a source to a destination host. In the Internet protocol suite this is maintained by the UDP and TCP protocols.

2.3.5 UDP

The User Datagram Protocol (UDP) runs on top of IP. The communication model is message oriented and stateless. Once a message is sent the sender forgets about the message. So if the message is dropped on the way, it will not be resent and the receiver will never receive it unless higher level applications ensures this

```go
package main

import ( "net" ; "fmt" ; "bufio" ; "strings" )

func handleConnection(conn net.Conn) {
  defer conn.Close()
  for {
    msg, err := bufio.NewReader(conn).ReadString('\n')
    if (err != nil) {
      fmt.Println("Error: " + err.Error())
      return
    } else {
      fmt.Print("From Client:", string(msg))
      titlemsg := strings.Title(msg)
      conn.Write([]byte(titlemsg))
    }
  }
}

func main() {
  fmt.Println("Listening for connection...")
  ln, _ := net.Listen("tcp", ":18081")
  defer ln.Close()
  conn, _ := ln.Accept()
  fmt.Println("Got a connection...")
  handleConnection(conn)
}
```

Figure 2.6 A TCP Server. We learn some more Go. The function `strings.Title` simply capitalises each word in the string. The `defer` keyword defers the function call until the enclosing function terminates. It is a good way to ensure that for instance connections are closed no matter what happens later: The deferred work is executed no matter how the function is exited, even if there is a panic. If you put the closing of connections at the end of the function, they would not be closed if you exit the function earlier in the code or the function terminated because of a panic.

themselves. Two things UDP does add compared to IP is checksums for data integrity and port numbers in order to multiplex the single channel between two machines offered by IP. A local service can register on a port with the OS to listen for packets on that port. Remote services can send packets to a specific port to have them arrive at the registered process. Multiple clients can send packets to the same port. Only one process can register locally at a port.

UDP ports are 16 bit numbers or equivalently integers between 0 and 65535. Port 0 is reserved, but is a permissible source port value if the sending process does not expect messages in response. The port numbers are partitioned into three ranges. Port numbers 1 to 1023 are used for standard services, for instance SSH is assigned to port 22 and HTTP is on port 80 by default. On ports 1024 to 49151 you can register your own protocol with the IANA organisation. For

```go
package main

import ( "bufio" ; "fmt" ; "net" ; "os" )

var conn net.Conn

func main() {
        conn, _ = net.Dial("tcp", "127.0.0.1:18081")
        defer conn.Close()
        for {
                reader := bufio.NewReader(os.Stdin)
                fmt.Print("> ")
                text, err := reader.ReadString('\n')
                if text == "quit\n" { return }
                fmt.Fprintf(conn, text)
                msg, err := bufio.NewReader(conn).ReadString('\n')
                if err != nil { return }
                fmt.Print("From server: " + msg)
        }
}
```

Figure 2.7 A TCP Client

instance Bitcoin is registered to port 8333. Ports 49152 to 65535 can be used for any purpose. Typically one picks a random port. Trying to pick a given hardcoded port is dangerous as it might already be taken. When writing an IP address and a port number together it is common to use the format `127.0.0.1:80`.

2.3.6 TCP

The Transmission Control Protocol (TCP) is another communication protocol run on top of IP. As opposed to UDP it provides a connection or stream-oriented communication model. The sender provides a continuous stream of bytes, which TCP moves to the receiver. TCP moves data between ports as does UDP. Moreover, TCP adds some level of reliability: it detects simple corruption of data along with loss or reordering of data. This is done by adding checksums and sequence numbers to packets. However TCP does not provide any authenticity, since any node through which the data is sent could change that data, and then just change the checksums and sequence numbers appropriately.

We assume that the reader has a rudimentary understanding of TCP, for example from a course on networks and operating systems.

2.3.7 Naming, DNS

We assume that the reader has a rudimentary understanding of Domain Name System (DNS), for example from a course on networks and operating systems. But for completeness we will recap the DNS system here. It is a decentralized

```
package main

import ( "net" ; "fmt" ; "strconv" )

func main() {
     addrs, _ := net.LookupHost("google.com")
     for indx, addr := range addrs {
              fmt.Println("Address number " + strconv.Itoa(indx) + ": " + addr)
     }
}
```

Figure 2.8 Let us look up the IP address of `www.google.com`.

```
package main

import ( "net" ; "fmt" ; "bufio" )

func main() {
     addrs, _ := net.LookupHost("www.google.com")
     addr := addrs[0]
     fmt.Println(addr)
     conn, err := net.Dial("tcp", addr+":80")
     if (conn!=nil) { defer conn.Close() }
     if (err!=nil) { panic(0) }
     fmt.Fprintf(conn, "GET /search?q=Secure+Distributed+Systems HTTP/1.1\n")
     fmt.Fprintf(conn, "HOST: www.google.com\n")
     fmt.Fprintf(conn, "\n")
     for {
             msg, err := bufio.NewReader(conn).ReadString('\n')
             if err != nil { panic(1) }
        fmt.Println(msg)
     }
}
```

Figure 2.9 Let us ask Google a question.

naming service which translates human readable names like `www.google.com` into IP addresses. This in turn allows one to contact for instance `www.google.com` via IP. There are 13 root servers. A root server is a server run by a designated organisation. The IP addresses of root servers are publicly known and rarely change, so that anyone can contact them. At the time of writing, these are assigned as follows.

a.root-servers.net	198.41.0.4	VeriSign, Inc.
b.root-servers.net	199.9.14.201	University of Southern California (ISI)
c.root-servers.net	192.33.4.12	Cogent Communications
d.root-servers.net	199.7.91.13	University of Maryland
e.root-servers.net	192.203.230.10	NASA (Ames Research Center)
f.root-servers.net	192.5.5.241	Internet Systems Consortium, Inc.

g.root-servers.net	192.112.36.4	US Department of Defense (NIC)
h.root-servers.net	198.97.190.53	US Army (Research Lab)
i.root-servers.net	192.36.148.17	Netnod
j.root-servers.net	192.58.128.30	VeriSign, Inc.
k.root-servers.net	193.0.14.129	RIPE NCC
l.root-servers.net	199.7.83.42	ICANN
m.root-servers.net	202.12.27.33	WIDE Project

If you own the domain `google.com` you can tell these servers what the IP address of `www.google.com` is. Then anyone else can go and ask them what the IP address of `www.google.com` is and for instance be told that it is `172.217.19.206`. Since there are few root servers and huge amount of domains and queries, root servers do not store the IP address of all domains. Instead, a root server will return to the user the IP address of another name server which keeps track of all addresses that end in `.com`. You might in fact get a whole list of IP addresses of such machines as there might be several for fault tolerance and load balancing. Then you go ask a random of these name servers for the IP address of `www.google.com`. It might in turn give you a list of IP addresses of name servers that keeps track of all addresses that end in `.google.com`. Then you ask a random of these name servers for the IP address of `www.google.com` and might finally be told `172.217.19.206`.

2.4 Programming Distributed Systems

It is notoriously hard to program distributed systems. Here are some vague advises which will be illustrated further throughout the book and programming exercises.

2.4.1 Local atomicity

Make sure the procedures running on one machine are `atomic`, i.e., if the procedure looks at local state, then make sure that local state does not change while the procedure executes. As an example, consider the following code:

```
func getMoney(c Caller, amount int) {
   if (amount <= account[c].amount) {
      sendMoney(c, amount);
      account[c].amount = account[c].amount - amount;
   }
}
```

This code allows a party to withdraw money if it has enough on its account. The `sendMoney` procedure is distributed and synchronous, it calls the system at the bank and waits to see whether the transfer actually succeeded. The problem here is that if customer BadGuy has 100 on its account and calls `sendMoney(BadGuy, 100)` twice, then both of them will succeed. First they check that $100 \leq 100$, which is the case. Then they both send 100. Then they both subtract 100 from the account of BadBuy, bringing it to -100. Even if BadGuy does not have full

```go
package main

import ( "net" ; "fmt" ; "bufio" ; "strings" )

func handleConnection(conn net.Conn) {
  defer conn.Close()
  myEnd := conn.LocalAddr().String()
  otherEnd := conn.RemoteAddr().String()
  for {
    msg, err := bufio.NewReader(conn).ReadString('\n')
    if (err != nil) {
      fmt.Println("Ending session with " + otherEnd)
      return
    } else {
      fmt.Print("From " + otherEnd  + " to " + myEnd + ": " + string(msg))
      titlemsg := strings.Title(msg)
      conn.Write([]byte(titlemsg))
    }
  }
}

func main() {
  ln, _ := net.Listen("tcp", ":18081")
  defer ln.Close()
  for {
      fmt.Println("Listening for connection...")
      conn, _ := ln.Accept()
      fmt.Println("Got a connection...")
      go handleConnection(conn)
  }
}
```

Figure 2.10 A multithreaded TCP Server. Try connecting multiple instances of the TCP client from Figure 2.7 to this server.

control of the scheduling of the two executions of `sendMoney` they are likely to be executed as above as `sendMoney` takes a long time to execute. The above is sometimes known as a reentrancy problem and sometimes as a race condition. They are worse in distributed systems as they could be actively exploited even if they are subtle. The way to avoid them is to keep things simple and make sure that each call to `getMoney` get executed fully before the next one is executed. On the Ethereum blockchain a very simple reentrancy bug famously cost 55 million USD and lead to the entire blockchain breaking into two.

2.4.2 Roll forward, don't rollback

If something goes wrong in one part of a system as part of one job, it might be tempting to erase the partial effect of the job. Say you told machine C to do some

```go
package main

import ( "fmt" ; "strconv" )

var sendernames = [5]string{"Alice", "Bob", "Chloe", "Dennis", "Elisa"}

var receivernames = [5]string{"Frederik", "Gary", "Hailey", "Isabel", "Jesper"}

func send(c chan string, myname string) {
     for i:=0; i<1000; i++ {
              // you send on a channel using <-
              c<- myname + "#" + strconv.Itoa(i)
     }
}

func receive(c chan string, myname string) {
     i:=0
     for {
              // you also receive from channel using <-
              msg := <-c
         fmt.Println(myname + "#" + strconv.Itoa(i) + " " + msg)
         i++
     }
}

func main() {
  c := make(chan string)
  for i:=0; i<5; i++ {
      go send(c, sendernames[i])
      go receive(c, receivernames[i])
  }
  // we do one blocking call to avoid that the program terminates
  receive(c, "Kacey")
}
```

Figure 2.11 A use of channels in Go. The purpose of the program is to illustrate the use of channels, the main tool for handling concurrency and parallelism in Go.

part of the job and told B to do some other part. Now B crashed. There are now two options. Rollback: you ask C to drop the work it did and you retry the whole job. Roll forward: you find some other way to finish the work B should have done. It can be notoriously hard to do a rollback. Maybe machine C already did its job and then crashed. Maybe C delegated the work to D, E and F, and E finished and crashed and D is still running but unresponsive because the network is slow. Trying a rollback can be an escalating logistic nightmare. Try to avoid it.

2.4.3 Don't rely on others

Don't rely on others doing their job correctly or in time, and don't rely on acknowledgements from others. In life, and in distributed systems. A common control structure in single machine programming is the synchronous function call.

If you are used to program with synchronous function calls it is easy to drag the pattern into distributed programming and program as follows: Machine A sends some values to machine B, then B does some work and returns a value or acknowledgement to A, at which point A knows the job was done and can proceed with its code. There are many problems with this programming pattern. If the acknowledge from B to A is lost, A will wait forever. If it is very late, so will A be. This can create a rippling effect of exceptions, and coding correctly with exception can be notoriously hard. This is even more the case in distributed systems, as you do not have a global view of what went wrong.

The most robust programming style for distributed systems is asynchronous message passing. If you need machine B to know something or do something, then you send it a message M and move forward with your own stuff. Don't wait for an acknowledgement from B. Don't even have the protocol include such a thing. Instead, program your system such that if B is down, then the job gets done anyway. Maybe there is a backup server that can take B's job. Maybe you have a background thread which will monitor the system and see if it shows the progress B should make, and if not it will occasionally resend M.

The above is the ideal. Oftentimes you cannot program like that, but always try it as the first thing. In Chapter 9 we will see Bracha Broadcast which is an excellent example of this programming style.

2.4.4 Failure is normal

Program as if failure is the normal. Or better, ensure that failure is the normal by having some process in your production environment which occasionally crash some of your servers and sends crazy messages. This is an emerging trend of chaos engineering. It is important to stress that the mistakes here are not part of testing, it is deliberately injected into the actual production environment by some process sometimes called a Chaos monkey. If you have a seperat evil minded department produce the Chaos monkey it will keep you programmers and designers on their toes. It is also discouraging for a potential attacker on the system. They cannot do anything to the system that it is not doing to itself on a daily bases.

2.4.5 Keep it simple and modular

This is the most important advice when programming distributed systems. If your program requires 200 lines of code, then they try to break it up into two logic components of 100 lines of code, which can each easily be implemented in a simple way and which have clear and simple dependencies and interactions. In distributed systems it pays off to think a lot before you touch the keyboard and start programming. Find the right abstractions and modules which will make everything simple. Big monolithic bundles of code are ensured to fail in distributed systems. The book has several examples of such simple abstractions, like broadcast, Byzantine agreement, and replicated state machines. We will for instance see how the very complex problem of replicating state machines is a simple application of broadcast and Byzantine agreement.

2.5 Concurrency

Concurrency is already hard when programming a multi-threaded process on a single machine. The problem becomes much harder in a distributed system, where delayed messages and crashed machines introduce events you cannot observe or foresee. We assume that the reader has an understanding of basic concurrency primitives likes Locks, MUTEX, Semaphores. You will also find these in Go. However, Concurrency in Go is inspired the theory of Communicating Sequential Processes (CSP)[5] and relies on threads synchronising via synchronising communication channels. This is a theory which handles concurrency by passing . Since distributed systems are all about message passing, then concurrency mechanism works well, and we encourage the reader to try to program with it to the fullest extend possible. Synchronising channels can easily be used to emulate MUTEX and Semaphores, but often a more elegant solution can be found. Try it out.

In Go calls on `Conn.Read` and `Conn.Write` are blocking. This holds for all programming languages, as these calls eventually are mapped to the equivalent calls in the OS, and these calls are blocking. For instance, this means that if you for want to receive data from two different TCP connections you cannot just make a loop where you alternate between reading from the two channels: if one of the channels run out of data, the call will block and prevent you from reading from the other channel. It *is* possible to set a timeout on a read and terminate the call after some time if no data is available. This is, however, an extremely cumbersome and error prone way to program. Instead, in programs where you have to handle several connections you want to use separate concurrent processes for doing this. In Go it is extremely easy to do concurrent programming, which is one of the reasons for choosing the language for our programming examples in this book. Consider the multithreaded TCP server in Fig. 2.10. In line 34 we added the `go` keyword in front of the call to `handleConnection`. This means that the call will be `non-blocking`. The current thread of execution will immediately terminate and therefore start the next iteration of the `for`-loop. At the same time a new thread of execution is created which starts executing `handleConnection`. These two threads will then proceed to run concurrently.

To coordinate concurrent threads Go uses two abstractions: MUTEX and channels. We assume the reader is familiar with using MUTEX (Mutual Exclusion) to coordinate concurrent threads. In Go a MUTEX has the type `Mutex`. It has two operations `Mutex.Lock()` and `Mutex.Unlock()`. These are reader/writer mutual exclusion locks which can be held by a single writer but several readers. They cannot be held by a writer and a reader at the same time. Take a look at the

```go
package main

import ( "fmt" ; "sync" )

type DNS struct {
    m map[string]string
    lock sync.RWMutex
}

func (dns *DNS) Set(key string, val string) {
    dns.lock.Lock()
    defer dns.lock.Unlock()
    dns.m[key] = val
}

func (dns *DNS) Get(key string) string {
    dns.lock.RLock()
    defer dns.lock.RUnlock()
    return dns.m[key]
}

func MakeDNS() *DNS {
    dns := new(DNS)
    dns.m = make(map[string]string)
    return dns
}

func GetAndSet(suf string) {
    for i:=0; i<10; i++ { dns.Set("X", dns.Get("X") + suf) }
    c<-0
}

var c = make(chan int)
var dns = MakeDNS()

func main() {
     go GetAndSet("1") ; go GetAndSet("2") ; go GetAndSet("3")
     <-c ; <-c ; <-c // wait for the three goroutines to end
     fmt.Println(dns.Get("X"))
}
```

Figure 2.12 A Go program with a race condition. See the caption of Figure 2.6 for an explanation on the use of the `defer` keyword.

`sync` package for more. The channel concept of Go is of more novelty. A channel has a type, which is the type of objects that can be sent over the channel. Any goroutine can send on a channel and any goroutine can receveive from a channel. It is safe for many Go-routines to send and receive at the same time without using MUTEXs. Notice that sending is *blocking*! So when a thread sends on a channel it blocks until the message is read. Also, if a thread reads from a channel, it will

```go
package main

import ( "fmt" ; "sync" )

type DNS struct {
    m map[string]string
    lock sync.Mutex
}

func MakeDNS() *DNS {
    dns := new(DNS)
    dns.m = make(map[string]string)
    return dns
}

func (dns *DNS) GetAndSetOnce(suf string) {
    dns.lock.Lock()
    defer dns.lock.Unlock()
    dns.m["X"]=dns.m["X"]+suf
}

func (dns *DNS) GetAndSet(suf string) {
    for i:=0; i<10; i++ { dns.GetAndSetOnce(suf) }
    c<-0
}

var c = make(chan int)

func main() {
    dns := MakeDNS()
    go dns.GetAndSet("1") ; go dns.GetAndSet("2") ; go dns.GetAndSet("3")
    <-c ; <-c ; <-c // wait for the three goroutines to end
    fmt.Println(dns.m["X"])
}
```

Figure 2.13 A Go program without a race condition

block until there is a value sent on the channel. This provides a very versatile way to both communicate and coordinate between threads. See Fig. 2.11 for a toy example. Try running the code. You will see that you get a panic at the end of the program as no more Go-routines are sending on the channel but still there are channels reading. Since sending and receiving are blocking, this means that the reading threads are in a deadlock: there are no threads left in the system which can make progress. Go can detect this and then causes a panic.

Beware that the build-in `map` type of Go is not thread safe, so if you have several Go-routines use the same `map`, you must use a Mutex. The map can handle one writer or multiple readers so it is sufficient to use an `RWMutex`. See Fig. 2.12 for an example of how to secure a `map` using an `RWMutex`.

In Fig. 2.12 the `map` has been correctly secured against multiple Go-routines, but we managed to make a bug in some other position. The intent of the program was that the printed string would contain 10 occurences of 1, 2, and 3. Yet, when one runs the program another result is printed. Even worse, the result differ from run to run. Here are some examples:

```
> go run concurrentmap.go
331111111111
> go run concurrentmap.go
333333333322222222221111111111
> go run concurrentmap.go
3333333333222221111111111
> go run concurrentmap.go
332222222222
> go run concurrentmap.go
333332222222222
> go run concurrentmap.go
331111111111
> go run concurrentmap.go
3333333333222222221111111111
```

Before reading the solution below, take a moment to see if you can realise what we did wrong? While you think, I will get coffee.—I assume you spotted it by now. The code

```go
dns.Set("X", dns.Get("X") + suf)
```

can be expanded to

```go
tmp := dns.Get("X")
tmp = tmp + suf
dns.Set("X", tmp)
```

at which point it is easier to see that between the call to `Get` and the call to `Set` some time will elapse. During that time the lock is released, so some other goroutine might read the same current value of `"X"`. Then the last of the these two Go-routines to call `Set` will overwrite to append done by the first one to call `Set`. What we really need to do is make the whole operation of updating atomic: no other processes should read or write while we do the update. In Fig. 2.13 is an example of how this could be done. Here some runs of the code:

```
> go run concurrentmap2.go
333333331111111111332222222222
> go run concurrentmap2.go
311111111113333333332222222222
> go run concurrentmap2.go
311111111113333333332222222222
> go run concurrentmap2.go
333333333311111111112222222222
```

```
> go run concurrentmap2.go
31111111111333333333322222222222
```

Note that we do not use the synchronized calls to `Get` and `Set`. The reason is that in Go a Mutex does not belong to a particular goroutine, so you can deadlock yourself if you try to lock the Mutex while you have the Lock. You are deadlocking because you are waiting for yourself to release the lock! This is different from some other programming languages and therefore something to watch out for if you are already familiar with concurrent programming in other languages. One thing to take away from the example is that it was not really the access to setting and getting on the map that needed to be thread safe. Concurrent programming is much harder than that: what you need is atomicity of the critical access to your data. This is also the reason why `map` is not already synchronised in Go: it is often not needed nor is it enough, so would it be to sit between two chairs!

Exercise 2.2 (putting the pieces together)

1. Write a server and client in Go which do the following.
   - When the client is started it prompts the user for an IP address and port number. Then it makes a TCP connection to the corresponding IP address and port. Then it will prompt the user for input until `quit` is typed. The input is sent to the server. Concurrently it reads lines of text coming from the server and prints them. Notice that several Go-routines may access `Conn.Read` and `Conn.Write` concurrently.
   - When the server is started it determines its local IP address and prints it. Then it start listening for connections on a random open port and prints the port number. It should be able to handle several concurrent connections. When the server receives a string on any incoming connections, it will prepend the IP address and port number of the sender to the string and send the string back to all its current connections.

   To get used to concurrency and channels in Go, use the following design for the server: each new connection is handled by a goroutine which reads the incoming lines of text. There is a single channel `outbound` of `string`, where the reading Go-routines send the strings they received after modifying them as specified. There is a single goroutine `broadcast` which is the single consumer of messages from `outbound`. It reads them iteratively and then sends them back to the clients.
2. Make a version of the above where the clients send messages automatically in a way similar to Fig. 2.11. Drop the upper bound in the loop such that the clients keep sending packages forever. With one server and ten clients, what kind of throughput do you get when running on one machine. For instance, how many packages can be sent to all clients in two minutes. If you have access to several physical machines try putting the clients on different machines. What kind of throughput do you get now?

```
package main

import ( "net" ; "fmt" ; "encoding/gob" ; "io" ; "log" )

type ToSend struct {
     Msg string // only exported variables are sent, so start the ...
     Number int // ... name of the fields you want send by a capital letter
}

func handleConnection(conn net.Conn) {
  defer conn.Close()
  msg := &ToSend{}
  for {
    dec := gob.NewDecoder(conn)
    err := dec.Decode(msg)
    if (err == io.EOF) {
      fmt.Println("Connection closed by " + conn.RemoteAddr().String())
      return
    }
    if (err != nil) {
      log.Println(err.Error())
      return
    }
    fmt.Println("From " + conn.RemoteAddr().String() + ":\n", msg)
  }
}

func main() {
  fmt.Println("Listening for connection...")
  ln, _ := net.Listen("tcp", ":18081")
  defer ln.Close()
  conn, _ := ln.Accept()
  fmt.Println("Got a connection from ", conn.RemoteAddr().String())
  handleConnection(conn)
}
```

Figure 2.14 A TCP server using Gob.

2.6 Marshalling

An important topic in communication in distributed systems is so-called marshalling. TCP allows to send a stream of octets a.k.a. bytes. Often we want to send more structured data, maybe even an object from an object oriented language like Go. This requires packing down the object as a stream of bytes and unpacking it at the other end. This is sometimes called serialization. This is extremely cumbersome and error prone to do "by hand". Fortunately modern programming languges offer mechanisms for doing it for you. To appreciate the trouble this save you, consider the so-called Big Endian-Little Endian Problem. When for instance a 32-bit integer is stored in memory, it is stored as four bytes. On some machines the most significant byte is stored first. On other machines the most significant

```
package main

import ( "bufio" ; "fmt" ; "net" ; "os" ; "encoding/gob" )

type ToSend struct {
     Msg string // only exported variables are sent, so start the ...
     Number int // ... name of the fields you want sent by a capital letter
}

func main() {
     ts := &ToSend{}
     conn, _ := net.Dial("tcp", "127.0.0.1:18081")
     defer conn.Close()
     for i:=0;; i++ {
        fmt.Print("> ")
             reader := bufio.NewReader(os.Stdin)
        m, err := reader.ReadString('\n')
        if err!=nil || m == "quit\n" { return }
        ts.Msg = m
        ts.Number = i
        enc := gob.NewEncoder(conn)
        enc.Encode(ts)
     }
}
```

Figure 2.15 A TCP client using Gob. Try removing some fields from the struct in the server, or adding some fields in the client, to appreciate the service Gob provides.

byte is stored last. So when an integer is sent as a stream of bytes over TCP, the sender and receiver need to explicitly agree on the order. The typical way to do it is using a network format. It is for instance once and for all agreed that the least significant byte is sent first. One advantage of this is that one can do serialization without knowing what kind of machine the data is sent to. This is a huge advantage as this might not be known until later and maybe the data is being sent to different types of machines.

Go has a serializability system called Gob. It is very simple yet very versatile. If you used a serialization system before, Gob will be familiar to you. In Go there is a notion of a field of a struct being exported or not. If the name of a field starts with a capital letter, it is exported. Otherwise it is not. Whether a field is exported or not impacts how it is visible in the package system. With respect to marshalling, one thing to keep in mind with Gob is that only exported fields are sent, so be sure to start those names with a capital if you want them sent on the network. A particularity of Gob is that it is very lenient when decoding. If you send an `int8` it will allow you to read it back into a `int16`. Also, if you send a `struct` with three fields, you can read it back into a struct with just two fields: names and types will be used to determine which fields to read. The last field is

```go
package main

import ( "fmt" ; "log" ; "net" ; "net/rpc" ; "os" ; "bufio" ; "time")

type PrintAndCount struct {
   x int
}

func (l *PrintAndCount) GetLine(line []byte, cnt *int) error {
   HardTask()
   l.x++
   fmt.Println(string(line))
   *cnt = l.x
   return nil
}

func HardTask() {
   time.Sleep(5 * time.Second)
}

func MakeTCPListener() *net.TCPListener {
   addy, err := net.ResolveTCPAddr("tcp", "0.0.0.0:42587")
   if err != nil { log.Fatal(err) }

   inbound, err := net.ListenTCP("tcp", addy)
   if err != nil { log.Fatal(err) }
   return inbound
}

func main() {
   // Register how we receive incoming connections
   go rpc.Accept(MakeTCPListener())

   // Register a PrintAndCount object
   rpc.Register(new(PrintAndCount))

   // Avoid terminating
   fmt.Println("Press any key to terminate server")
   bufio.NewReader(os.Stdin).ReadLine()
}
```

Figure 2.16 An RPC server

just ditched. This will allow to read old serialized data later even if a protocol had its data updated to have new fields. It will also allow old and new versions of protocols to run at the same time even if packets have new optional fields added.

```go
package main

import ( "log" ; "net/rpc" ; "bufio" ; "os" ; "fmt")

func main() {
   client, err := rpc.Dial("tcp", "localhost:42587")
   if err != nil { log.Fatal(err) }

   in := bufio.NewReader(os.Stdin)
   for {
      line, _, err := in.ReadLine()
      var reply int


      // Synchronous call
      err = client.Call("PrintAndCount.GetLine", line, &reply)

      if err != nil { log.Fatal(err) }
      fmt.Println("Strings printed at server: ", reply)
   }
}
```

Figure 2.17 A synchronous RPC client

2.7 Remote Procedure Calls

With serialization in place we can conveniently move objects over the network. The next step is to add some session control. A typical control pattern in programming is the function call: some values are sent to a sub-routine and some return values are returned. This pattern also occurs naturally in distributed programming: some values are sent to a server (using serialization) and later some return values are sent back (using serialization). Remote Procedure Calls (RPC) set up standard routines and language constructions for doing this. It works as follows:

1. The parameters are sent over the network to the remote machine using serialization.
2. The function is run on the remote machine and some return value is produced.
3. The return values are sent over the network to the caller using serialization.

In object oriented programming languages, RPC is also sometimes known a Remote Method Invocation (RMI). The only difference is that when the function is run on the remote machine, the function may be a method of an object, and that method may in turn modify the state of that object.

In Fig. 2.16 and Fig. 2.17 you can find an example of how to use RPC in Go. Try to run it to make sure it works on your machine. Try to connect to the same server with several clients and see what happens. Notice that the code is not thread safe, as we have used no concurrency control whatsoever. Some things to notice:

```
package main

import ( "log" ; "net/rpc" ; "bufio" ; "os" ; "fmt")

func PrintWhenReady(call *rpc.Call) {
   <-call.Done // this channel unblocks when the call returns
   if call.Error != nil { log.Fatal(call.Error) }
   fmt.Println("Strings printed at server: ", reply)
}

var reply int

func main() {
   client, err := rpc.Dial("tcp", "localhost:42587")
   if err != nil { log.Fatal(err) }

   in := bufio.NewReader(os.Stdin)
   for {
      line, _, _ := in.ReadLine()

      // Asynchronous call
      call := client.Go("PrintAndCount.GetLine", line, &reply, nil)
      go PrintWhenReady(call) // Handles the reply when ready

      fmt.Println("See, I can still do stuff!")
      fmt.Println("See, I can still do stuff!")
      fmt.Println("OK, I'm bored...")
   }
}
```

Figure 2.18 An asynchronous RPC client

- You make an object available on the network by registering it as shown. You can only register one object of each given type as it is accessed via the type name.
- For a method of a registered object to be visible to the clients they need to take two arguments and have return type `error`. The second argument must be a pointer. The first argument is the input. The second input is used to hold the output.
- Both arguments must be exported (or builtin) types.

The client in Fig. 2.17 makes a synchronous call: it blocks until the reply returns. It is also possible to make an asynchronous call where the RPC does not block. In that case you set up a goroutine that waits for the reply. See Fig. 2.18 for an example.

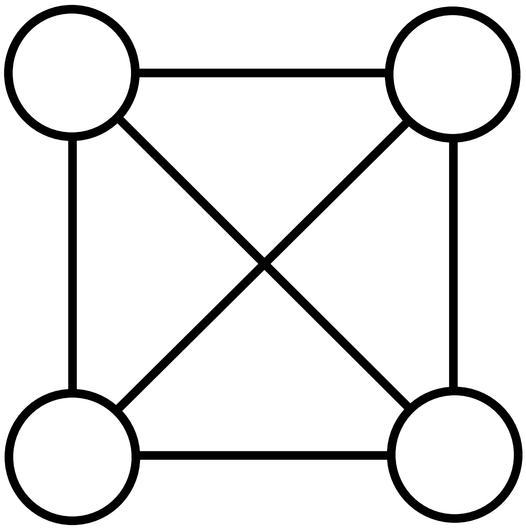


Figure 2.19 A fully connected system with four servers.

2.8 Logically Organising Connections

With the Internet in place we are now in a situation where all machines in principle can make connections to all other machines. However, maintaining a connection consumes resources in the OS so it is not viable to keep a connection open to a large number of other machines. In this section we take a brief look at the ways connections and machines can be organized.

2.8.1 Fully Connected

The simplest solution in a distributed system is to let everyone in the system make a connection to everyone else in the system. This is often done in small systems where for instance a server is replicated. However, with n servers, the number of connections is in the order of n^2. In practice this does not scale well for larger system. However in some cases it might be necessary: for instance, in systems where you want to tolerate Byzantine errors, it might not be an option to send communication via other nodes (which might be corrupted). Therefore a fully connected network is often used for small fault tolerant systems. We will in later sections study how to achieve Byzantine agreement in such systems.

2.8.2 Client-Server

Probably the most widespread system architecture is the client-server model. There is a single server to which all clients connect. All information exchanged between clients is exchanged via the server. Platforms like Dropbox, Amazon, most gaming platforms, and whatnot are organised this way. An obvious prob-

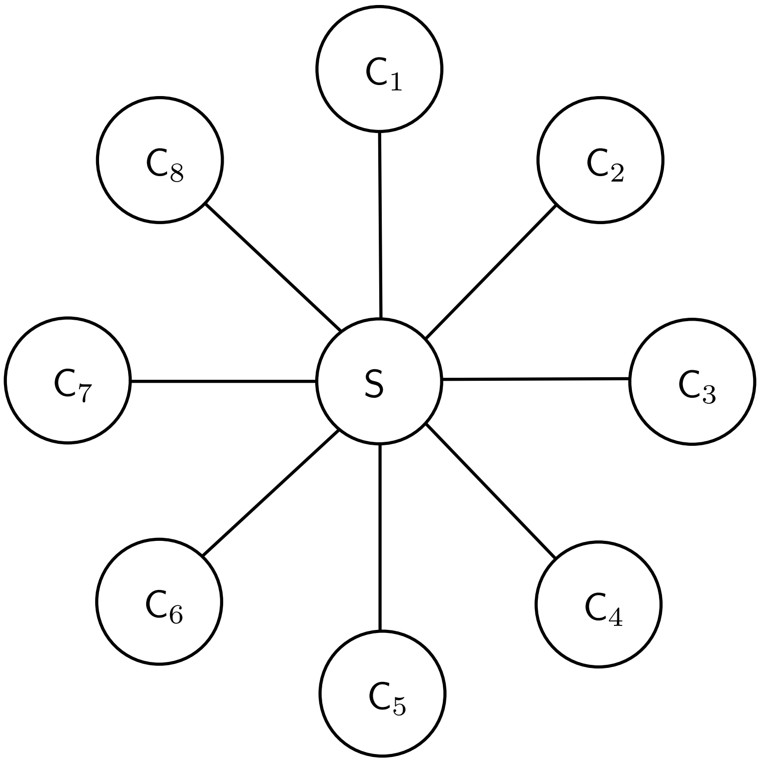


Figure 2.20 A client-server system.

lem with the plain client-server architecture is that it has a single point of failure/attack/trust. If the server crashes/is attacked/is corrupted the security of the system breaks down.

2.8.3 Client-Replicated Server

The obvious way to deal with the single point of attack of the client-server model is to replicate the server. This also creates the possibility to take some load off the server if it serves a huge number of clients. This is the most common architecture on the Internet. It opens up a lot of problems with keeping the copies of the replicated server consistent. We will spend a lot of time later understanding how to solve these problems.

2.8.4 Edge Nodes

When having a replicated server, it is common to put it on a protected network and let all accesses be mediated by so-called edge nodes. Clients contact the edge nodes, which will then distribute the workload over the replicates. When we later

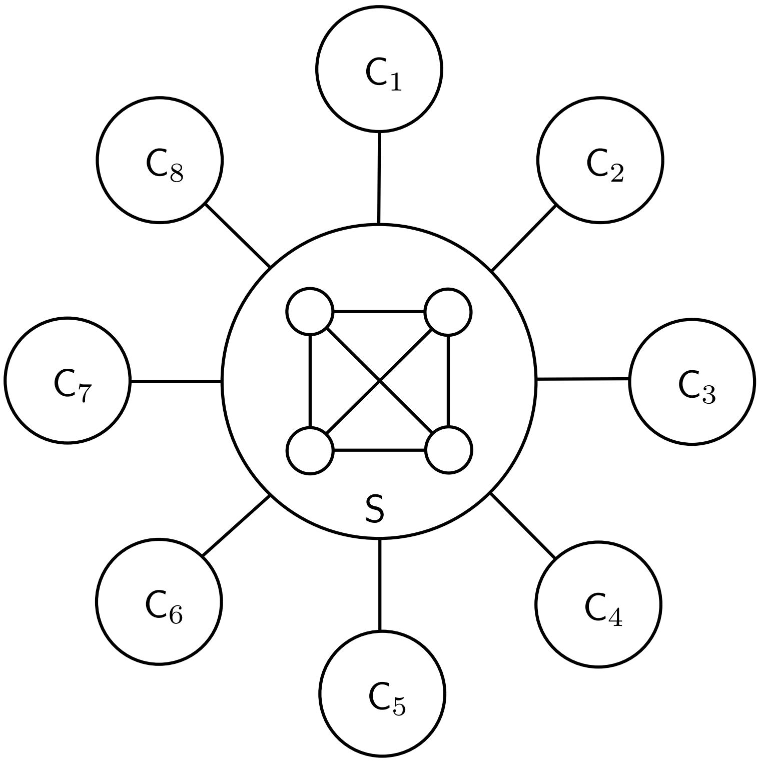


Figure 2.21 A Replicated Server

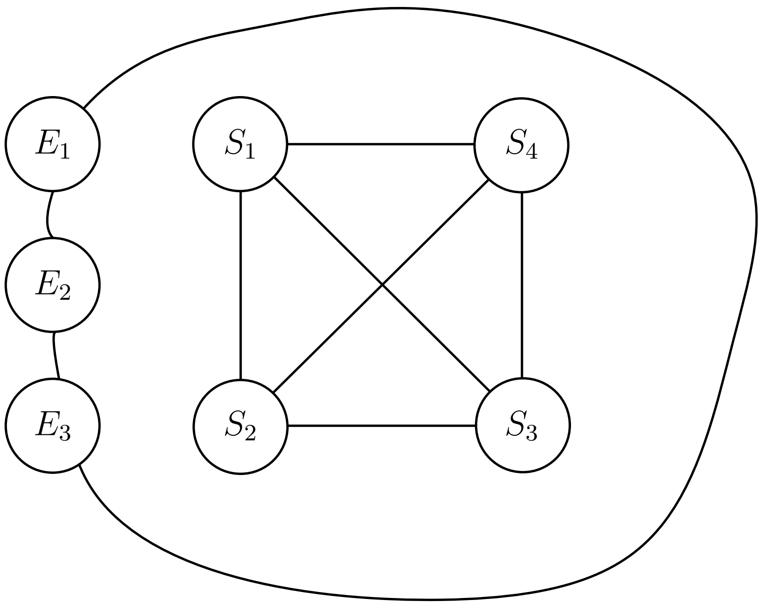


Figure 2.22 Replicated Server with Edge Nodes

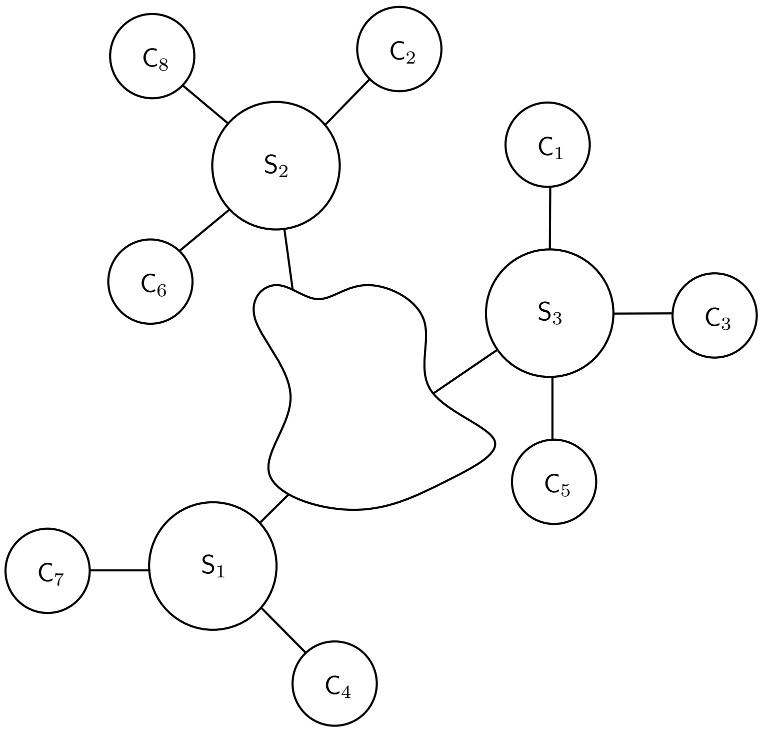


Figure 2.23 Illustration of organization of SMTP

talk about Firewalls we will return to how this model can increase security by protecting the servers against direct contact with evildoers.

2.8.5 SMTP

As our last example of how to organize communication in a structured way, we consider the Simple Mail Transfer Protocol (SMTP). Each domain, like `au.dk` has its own mail server. The IP address of the mail server is stored in the DNS system. Here is an example of how to look up the IP address of the mail server of Aarhus University using the `dig` command on Linux. First we look up the DNS name of the server:

```
> dig au.dk MX
;; ANSWER SECTION:
au.dk. 26992 IN MX 10 au-dk.mail.protection.outlook.com.
```

The `MX` indicates that we look for the main server. We see that the name is `au-dk.mail.protection.outlook.com`. We then use an `A` to look up the IP address of that DNS name:

```
> dig au-dk.mail.protection.outlook.com A
;; ANSWER SECTION:
au-dk.mail.protection.outlook.com. 15 IN A 213.199.154.234
au-dk.mail.protection.outlook.com. 15 IN A 213.199.154.170
```

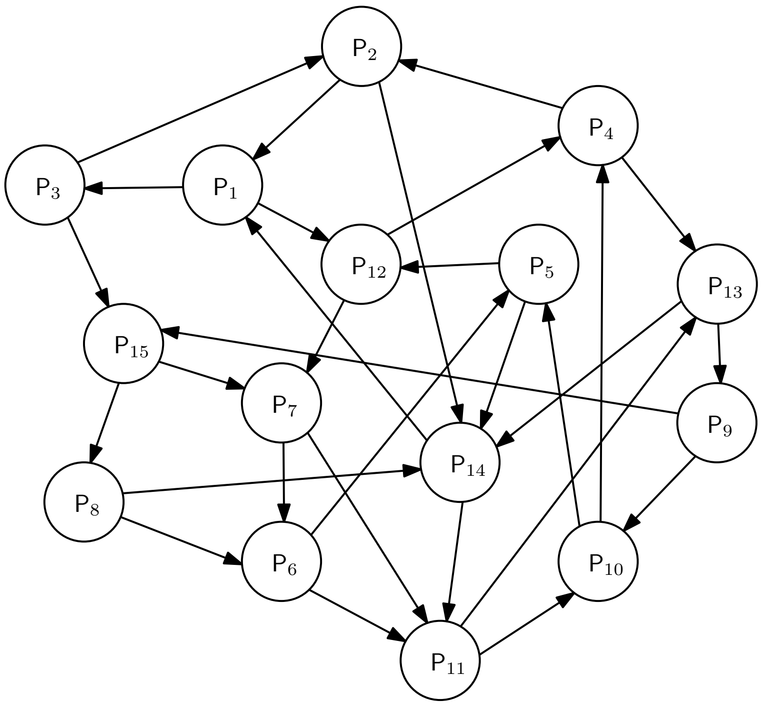


Figure 2.24 Random Peer-to-Peer

There are two addresses returned indicating that the server is indeed replicated. They might also be edge nodes with more machines behind them.

If your are at Aarhus university and send an e-mail, your mail program might send it to one of these servers that receive it for you initially. These local mail servers can then forward the e-mail to the mail server of the receiving institution. This might take several attempts if these servers are down or slow. Your mail client program is not involved in these attempts at re-sending the message, which are solely handled by the mail server. The system architecture is illustrated in Fig. 2.23.

2.9 Unstructured Peer-to-Peer

A common theme in large distributed systems is to use peer-to-peer networks. All parties will not keep a connection to all other parties: it is expensive to keep a large number of open connections, and sometimes parties are not even aware of all other parties in the network. There are two main kinds of peer-to-peer networks, namely structured peer-to-peer networks and unstructured peer-to-peer networks. In structured peer-to-peer networks, the connections between nodes are established in a deterministic way. In unstructured overlay networks the connections are formed at random. In Fig. 2.24 a random peer-to-peer network is shown, where each party has chosen two other parties to connect to at random.

A peer-to-peer graph can be used to disseminate messages. If you want to send a message, simply send it to the peers you have chosen. If a node receives a message it *has not see before* then it recursively disseminate it the same way. Consider Fig. 2.24 and consider a situation where P_3 sends a message. Assuming that each message transfer takes exactly the same time, then the message would propagate in the following way:

$$\{\mathsf{P}_3\}$$

$$\{\mathsf{P}_3, \mathsf{P}_2, \mathsf{P}_{15}\}$$

$$\{\mathsf{P}_3, \mathsf{P}_2, \mathsf{P}_{15}, \mathsf{P}_1, \mathsf{P}_{14}, \mathsf{P}_8, \mathsf{P}_7\}$$

$$\{\mathsf{P}_3, \mathsf{P}_2, \mathsf{P}_{15}, \mathsf{P}_1, \mathsf{P}_{14}, \mathsf{P}_8, \mathsf{P}_7, \mathsf{P}_{12}, \mathsf{P}_{11}, \mathsf{P}_6\}$$

$$\{\mathsf{P}_3, \mathsf{P}_2, \mathsf{P}_{15}, \mathsf{P}_1, \mathsf{P}_{14}, \mathsf{P}_8, \mathsf{P}_7, \mathsf{P}_{12}, \mathsf{P}_{11}, \mathsf{P}_6, \mathsf{P}_4, \mathsf{P}_{13}, \mathsf{P}_{10}, \mathsf{P}_5\}$$

$$\{\mathsf{P}_3, \mathsf{P}_2, \mathsf{P}_{15}, \mathsf{P}_1, \mathsf{P}_{14}, \mathsf{P}_8, \mathsf{P}_7, \mathsf{P}_{12}, \mathsf{P}_{11}, \mathsf{P}_6, \mathsf{P}_4, \mathsf{P}_{13}, \mathsf{P}_{10}, \mathsf{P}_5, \mathsf{P}_9\}$$

Two important measures of how good a peer-to-peer graph is are:

Diameter For each pair $(\mathsf{P}_i, \mathsf{P}_j)$ of parties, define their distance to be the shortest path from P_i to P_j. If they are not connected, then $\mathsf{Distance}(\mathsf{P}_i, \mathsf{P}_j) = +\infty$. Then the diameter of the graph is

$$\mathsf{Diameter} = \max_{\mathsf{P}_i, \mathsf{P}_j} \mathsf{Distance}(\mathsf{P}_i, \mathsf{P}_j) \ .$$

To compute the distance from two parties, one can flood the network starting from P_i and see how long it takes to reach P_j, as we did above for P_3. From the above example we can see that the distance from P_3 to P_9 is 5, so the diameter of the above graph is at least 5.

Connectivity If the diameter of a graph is $+\infty$, then we say that the network is unconnected. When it is unconnected, there are two parties between which there is no path. The peer-to-peer network then does not allow these parties to communicate. The connectivity of a graph is defined to be the minimal number of edges that must be removed to make the graph unconnected.

The reason why we like to use random peer-to-peer graphs is that they are easy to form and maintain and at the same time they have low diameter and high connectivity on average.

Exercise 2.3 (diameter, connectivity) Compute the diameter and connectivity of the graph in Fig. 2.24.

We now briefly investigate the connectivity of a random graph. In the graph from in Fig. 2.24, each node is connected to two other nodes. We then say that the out-degree is 2.

Theorem 2.1 *Assume that $n \geq 25$ and $\kappa \geq 2$. If you pick a random peer-to-peer graph on n parties with out-degree κ, then the probability that it is not connected is at most $2^{-\kappa}$.*

Let P be the set of all n parties. We call a subset $S \subset P$ non-trivial if $S \neq \emptyset$ and $S \neq P$. We call a non-trivial subset $S \subset P$ a cut if there is no path from a party in S to a party in $P \setminus S$. Notice that a graph is unconnected if and only if it has a cut.

We now fix a non-trivial subset S and compute the probability that S becomes a cut if a random peer-to-peer network of out-degree κ is chosen. Let $s = |S|$. Each of these s parties will pick κ random peers to connect to. For S to be a cut, all these $s\kappa$ connections need to end up in S. If $s \leq \kappa$, then the probability of this is 0, as each of the parties in S will pick at least one peer from outside s. So assume for the rest of the analysis that $s > \kappa$. The probability that all peers are picked inside S is less than

$$\left(\frac{s}{n}\right)^{s\kappa} .$$

The right-hand side is the number we get if a peer picks the connections with possible repetition and allows might pick itself. It will not pick itself and will not pick the same connection twice. But this just makes the probability of getting a connection outside S higher.

There are at most 2^n different subset S to consider, as there are at most 2^n subsets of P. So by a union bound, the probability there is a cut is at most

$$2^n \max_{s>\kappa} \left(\frac{s}{n}\right)^{s\kappa} .$$

We have that

$$2^n \max_{s>\kappa} \left(\frac{s}{n}\right)^{s\kappa} = \max_{s>\kappa} \left(2^{n/\kappa} \left(\frac{s}{n}\right)^{s}\right)^{\kappa} ,$$

so it is sufficient to prove that

$$\max_{s>\kappa} 2^{n/\kappa} \left(\frac{s}{n}\right)^{s} \leq 1/2 .$$

This is the same as proving that

$$n/\kappa + \max_{s>\kappa} \log_2 \left(\left(\frac{s}{n}\right)^{s}\right) \leq -1 ,$$

or

$$n/\kappa \leq \max_{s>\kappa} s \log_2 \left(\frac{n}{s}\right) - 1 .$$

The expression $s \log_2 \left(\frac{n}{s}\right)$ is maximal at $s = n/e$, where it is

$$(n/e) \log_2 e > 0.54n .$$

So, we need that

$$n/\kappa \leq 0.54n - 1 .$$

If $\kappa \geq 2$, then this holds for all $n \geq 25$.

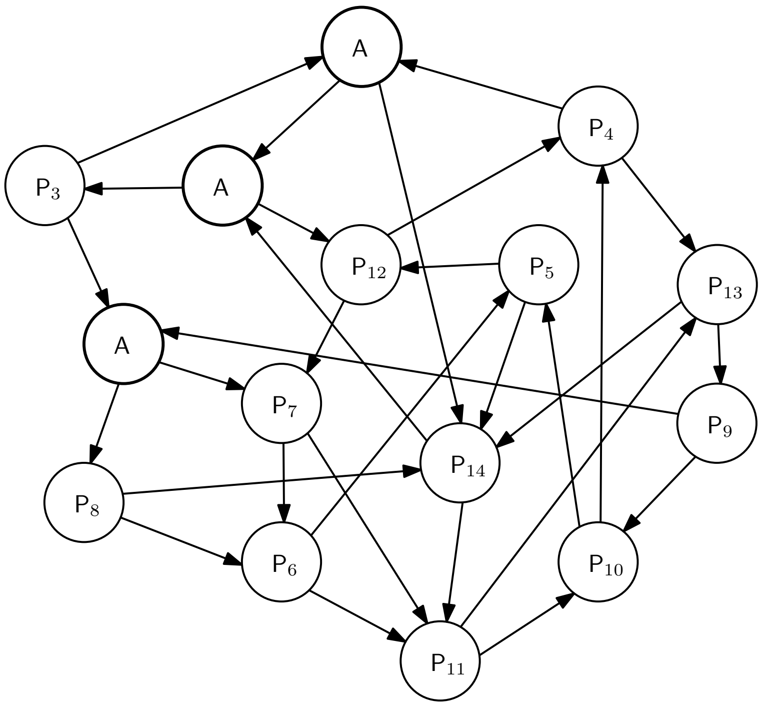


Figure 2.25 An Eclipsing Attack

2.9.1 How to Build and Maintain

To build a random peer-to-peer network, each node could hold a list of all other nodes that are present in the network (we call this its network list), and then pick its connections at random from this list. When a new node enters the network, it will flood its presence, for instance its IP address, and all other nodes will add the new node to their network list. If a node becomes unresponsive, the other nodes will remove it from their network list. To enter the network initially, a node needs to know at least one other node of the network and ask it for its network list. In practice it is not possible to have a list of every node on the network, so there are other methods to build a random looking network. Each node only holds a partial view. When entering the network you get the partial view of your entry point. Then you can contact some of the known peers to get their partial view and that way learn about more peers, and yourself build a random looking partial view of the system.

2.9.2 Gossiping

Sending a large message M in a flooding network can be expensive. Each node sends and receives it many times. Here is a variant sometimes known as gossiping. It brakes the distribution of M up into a push and pull phase. You send M to your neighbours in sequence with a small break in between. Each time you send it the neighbour reports back if it already had M. After some number, say 10, neighbours reported that they already had the message, you take this as a signal that most nodes have the message now and stop sending it. This was the push phase. This might by chance result in some nodes not getting M. Therefore each node will occasionally pull its neighbours to check if they have new messages.

Another way to reduce the overhead of flooding is to flood only a short notification that there is some large update. Then each client will pull its neighbours for it. It will stop pulling after getting M once. Now it can reply to pull requests from its own neightbours. This can be much slower, but it ensures that each M is received only once per client.

2.9.3 Eclipsing Attacks

A dangerous type of Byzantine attack on peer-to-peer networks is called eclipsing attacks. Assume that the adversary controls some of the nodes $A \subset P$ of the network. If removing the connections of all these nodes creates a cut with no connections in either direction, we say that the adversary has successfully mounted an eclipsing attack, since the adversary has managed to divide the network into two groups. All communication between the groups goes via the adversary, who can therefore control all information. This type of attack is easier to perform on a single party. For an example, consider Fig. 2.25, where P_3 has been eclipsed by an adversary A controlling only three nodes.

A particularly easy way to mount an eclipsing attack is for an adversary to create many copies of itself. It might only have a single machine, but might create millions of IP addresses and add them to the peer-to-peer system. This is called a Sybil attack. If the system only has 10,000 honest nodes but 1000,000 IP adresses pointing to the adversary, only one in a hundred of the IP addresses in a random selection would be to honest parties. So even a node picking a hundred random peers would with good probability be eclipsed. To avoid Sybil attacks real world peer-to-peer networks apply a number of heuristics when selecting the connections. One for instance does not pick multiple IP adresses from the same sub-domain / region.

It is particularly easy to be a victim of Eclipsing attacks when first entering a network. One needs to be sure to ask at least one honest node about its network list.

Exercise 2.4 (implement a toy peer-to-peer network) This exercise asks you to program in Go a toy example of a peer-to-peer flooding network for sending strings around. The peer-to-peer network should then be used to build a distributed chat room. The chat room client should work as follows:

1. It runs as a command line program.
2. When it starts up it asks for the IP address and port number of an existing peer on the network. If the IP address or port is invalid or no peer is found at the address, the client starts its own new network with only itself as member.

3. Then the client prints its own IP address and the port on which it waits for connections.
4. Then it will iteratively prompt the user for text strings.
5. When the user types a text string at any connected client, then it will eventually be printed at all other clients.
6. Only the text string should be printed, no information about who sent it.

The system should be implemented as follows:

1. When a client connects to an existing peer, it will keep a TCP connection to that peer.
2. Then the client opens its own port where it waits for incoming TCP connections.
3. All the connections will be treated the same, they will be used for both sending and receiving strings.
4. It keeps a set of messages that it already sent. In Go you can make a set as a map `var MessagesSent map[string]bool`. You just map the strings that were sent to true. Initially all of them are set to false, so the set is initially empty, as it should be.
5. When a string is typed by the user or a string arrives on any of its connections, the client checks if it is already sent. If so, it does nothing. Otherwise it adds it to `MessagesSent` and then sends it on all its connections. (Remember concurrency control. Probably several go-routines will access the set at the same time. Make sure that does not give problems.)
6. Whenever a message is added to `MessagesSent`, also print it for the user to see.
7. Optional: Try to ensure that if clients arrive on the network after it already started running, then they also receive the messages sent *before* they joined the network. This is not needed for full grades.

Add this to your report:

1. Test your system and describe how you tested it.
2. Argue that your system has eventual consistency in the sense that if all clients stop typing, then eventually all clients will print the same set of strings.

2.10 Structured Peer-to-Peer

There is also a notion of structured peer-to-peer networks. A popular structured peer-to-peer network is the Chord network. Here each node has an identity, for instance its IP address. This identity is mapped deterministically into an integer between 0 and $B - 1$ for some bound B. It is not important what B is exactly, a long as it is so large that the chance that two different nodes are mapped onto the same integer is small. We can imagine all the numbers from 0 to $B - 1$ being wrapped around a circle with 0 at the top and $B - 1$ being the last point before we hit 0 again. If we kept mapping the integers like this B would hit the same point as 0, $B + 1$ would hit the same point as 1 and so on.

Now when a number of nodes with identities $A, B, C, \ldots$ enter the network, they will each map their identity into a number. We then imagine them sitting at the corresponding point on the circle. Each node will then make a connection to the node that is 1 step away on the circle, make a connection to the node that is 2 steps away, and the node 4 steps away, 8 steps away, 16, 32, and so on. If there is no node at those exact points they make a connection to the next node on the circle. So in Fig. 2.26, node A would make a connection to M, I, H, and N. And node N would have connections to F, E, K, G and M. To send a message to a recipient, a node will send it to the closest known peer before the recipient. That peer will then send it along until it reaches the recipient. If for instance A wants to send a message m to G, then it sends it to N and then N sends it to G. The particular way the connections have been chosen ensures that the diameter of the graph is only $\log_2(B)$, so it does not take too long to send a message.

Having a structured network also allows to use it as a shared variable space. Say each position i in the ring corresponds to a variable V_i. The node N_i closest to position i can be made responsible of keeping V_i. To read V_i pull it from N_i. To write it, push the update to N_i.

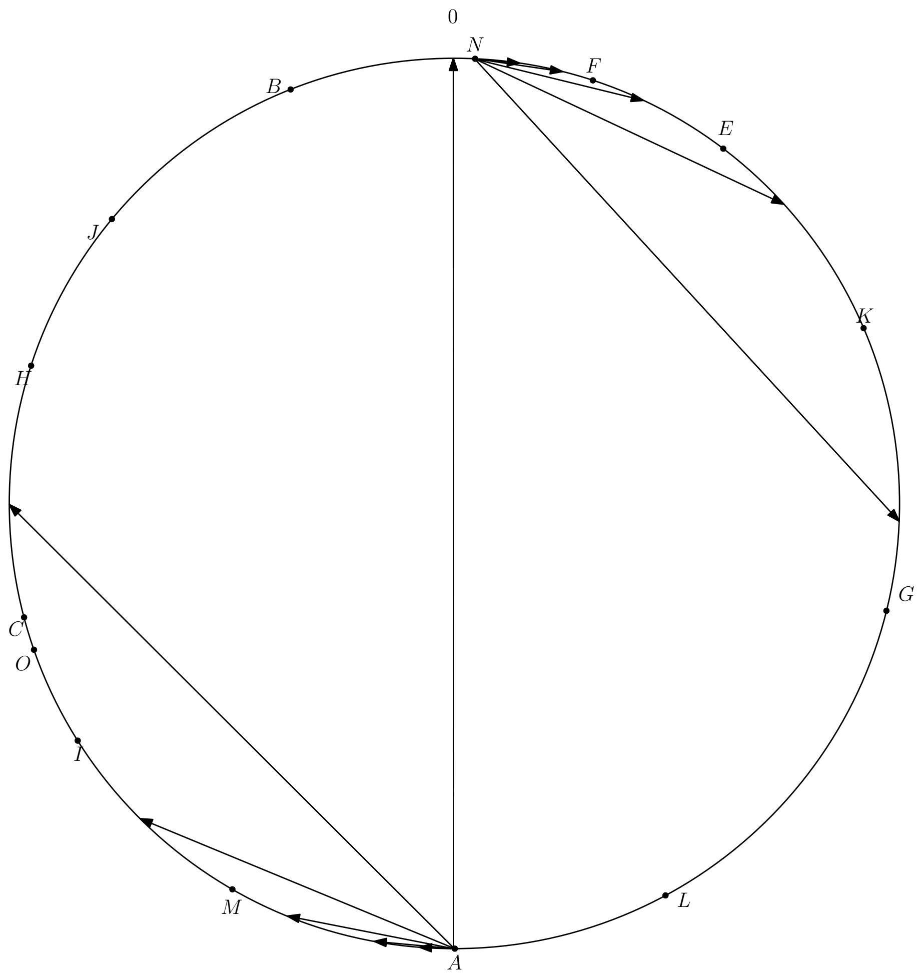


Figure 2.26 Chord Illustration

3

A Model of Distributed Systems

Contents

In this book we would like to prove some results about distributed systems. To be able to prove any statement about any object, both the statement and the object need to be formalised with mathematical precision. For this a so-called system model is used, where all entities are modelled within some formal framework. In the following we will first formalise a basic framework and then show how to use it to model processes, channels, disks and other system components.

In most of the book we will only sketch proofs and will not go to the level of detail of the model given in this chapter. This chapter serves as a basis for the formally inclined reader to learn how the informal proofs in other parts of the book might be brought on more precise grounds.

3.1 Basic System Model

3.1.1 Interactive Agents

The most basic entity will be an Interactive Agent (IA) (see an example in Fig. 3.1), which is formally defined by a tuple $(\mathcal{P}, \sigma_0, T)$ where:

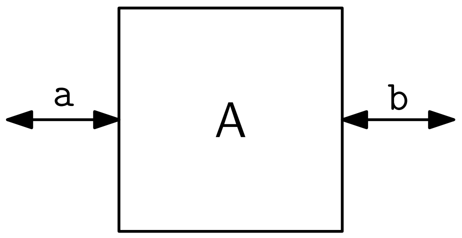


Figure 3.1 An Interactive Agent A with $\mathcal{P} = \{\mathtt{a}, \mathtt{b}\}$. Let $\sigma_0 := 0$. We can define $T(\sigma, P, m) = (\sigma', Q, m')$. If m is not an integer, then let $Q = P$ and let $m' = \perp$ to signal error. Otherwise, let $\sigma' = \sigma + m$ and let $Q = P$ and $m' = \sigma'$. This is then an interactive agent which keeps an accumulated sum of the integers input on its ports. When you send on some port a new number to add to the accumulator, you are returned the accumulated sum on the same port.

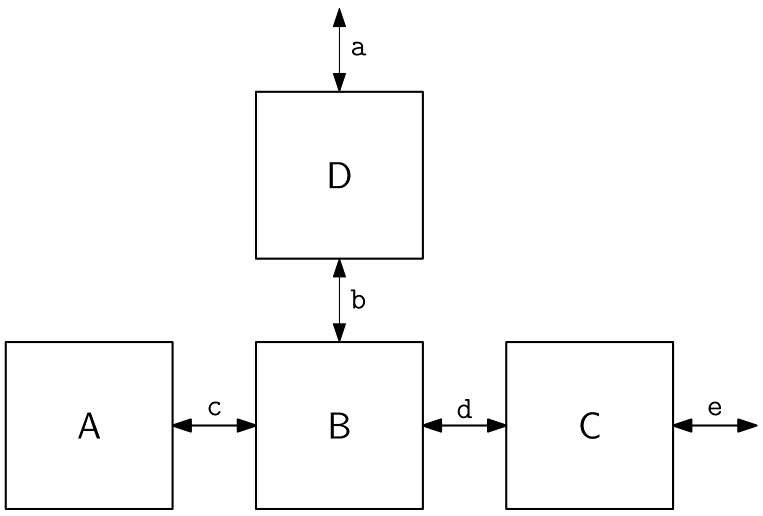


Figure 3.2 An Interactive System with open ports a and e

- $\mathcal{P}$ is a finite set containing the names of the ports on which the IA can receive inputs and outputs. Each port can be used both for sending and receiving.
- σ_0 is the initial state of the IA.
- T is the transition function. The transition function T says what the system does in response to a message being received on one of the ports. It depends on the current state σ (initially σ_0), the port name $P \in \mathcal{P}$ and the message m being received. The output of T specifies the new current state σ', a new port $Q \in \mathcal{P}$ and some new message $m' \in \mathcal{P}$. We write $(\sigma', Q, m') \leftarrow T(\sigma, P, m)$. Notice that

by this rule the transition function must always send a new message on some legal port.

We think of the IA A as a box holding its current state σ. We will therefore sometimes write $(Q, m') \leftarrow A(P, m)$ to mean: fetch the current state σ from A, compute $(\sigma', Q, m') \leftarrow T(\sigma, P, m)$ and replace the current state in A by σ'. We allow the transition functions to be randomised (therefore, they are not strictly "functions" in a mathematical sense). In this case the value $T(\sigma, P, m)$ is a random variable over values of the form (σ', Q, m'). When we write $(\sigma', Q, m') \leftarrow T(\sigma, P, m)$ we mean that (σ', Q, m') is sampled from the random variable $T(\sigma, P, m)$. This allows to model an algorithm which for instance samples a random RSA key (n, e, d), saves (n, d) in the state σ' and sends (n, e) on some port. An RSA key is a key used for public-key encryption and signature schemes, which we discuss in detail in Chapter 5 and Chapter 6.

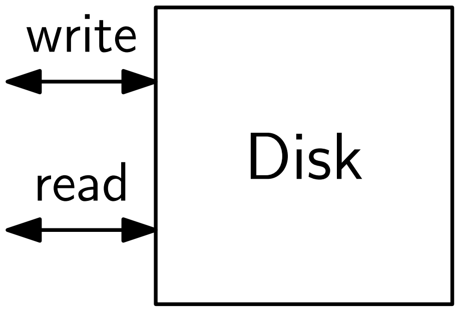
Even though T is not formally a function we will still call it the transition function. We make the requirement that the transition function does not depend on the specific port names, i.e., if we renamed all port names in a unique manner the IA would still behave the same modulo the new port names. The reason for this is that if we later want to compose two systems with conflicting port names, then we can just rename the port names in one system without changing its input-output behaviour.

3.1.2 Interactive Systems

An interactive system (IS) S is just a set of IAs. If there are two IAs with the same port name, then we think of those as being connected.[1] If there are more than two IAs with the same port name, then we say that the system is malformed and we write $S = \perp$. When we have two systems S_1 and S_2 we compose them simply be taking set union $S_1 + S_2 = S_1 \cup S_2$. It is easy to see then that if S_1, S_2, S_3 are interactive systems then $S_1 + (S_2 + S_3) = (S_1 + S_2) + S_3$ and $S_1 + S_2 = S_2 + S_1$. The ports not connected to other ports are called open ports. An IS can be activated on an open port P by sending some message m. We say that we input (P, m) to S. The execution proceeds as follows. Let A_0 be the IA with port P, let $m_0 = m$ and $P_0 = P$. Then we compute $(P_1, m_1) \leftarrow A_0(P_0, m_0)$. If P_1 is a closed port, then let A_1 be the other IA with port P_1. Then compute $(P_2, m_2) \leftarrow A_1(P_1, m_1)$ and so on until we compute $(P_{t+1}, m_{t+1}) \leftarrow A_t(P_t, m_t)$ and P_{t+1} is an open port. Then let $Q = P_{t+1}$ and $m' = m_{t+1}$. Return (Q, m'). Let $\mathcal{P}$ be the open ports of S. Let the internal state of S be the internal state of all the IAs composing the S. Notice that this makes an IS a function from pairs (P, m) with $P \in \mathcal{P}$ to pairs (Q, m')

[1] We could in the IA formalizm have given the ports types and asked that the types of connected ports match. However, if we start putting types on IAs, we would probably have to add so-called session types to get a useful framework. A simple notion of typed ports seem to serve no real purpose, sitting between two chairs. So, we have chosen to not treat the issue with types in this book. The reader who worries about types can take the type of all ports to be the set of all finite bit-strings. All the messages being sent around are then represented as bit-strings in some canonical encoding.

with $Q \in \mathcal{P}$. In computing the output the IS also updates the internal state of the system. So an IS is in some sense just a more structured IA. There are some formal differences though. For instance, an interactive system might loop forever, whereas we require that IAs always terminate.



Ports There are two ports write and read.
Init When the disk is initialised it creates a map $M : A \to B$ between sets A and B. We let $\perp$ denote that an entry is uninitialised and we assume that $\perp \in B$. Initially set $M[x] = \perp$ for all $x \in A$.
Write When it receives a value $(x, v) \in A \times B$ on write it sets $M[x] = v$ and returns DONE on write.
Read When it receives a value $x \in A$ on read it returns $M[x]$ on read.

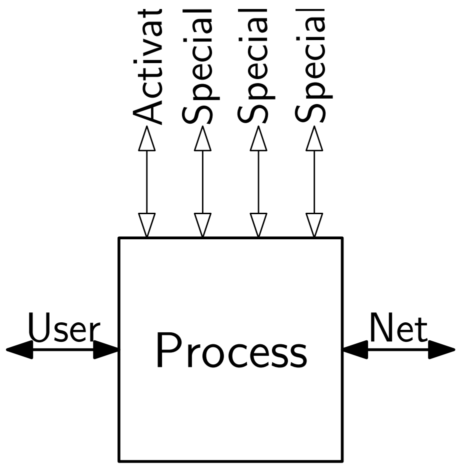
Figure 3.3 A Disk.

3.2 Modelling Persistent Storage

An important tool in fault tolerant distributed systems is some kind of permanent storage that will survive a crash. We will sometimes call it a disk and sometimes a log, or just persistent storage. We give an oversimplified model of persistent storage in Fig. 3.3. Implementing it in practice is a whole science in itself. What is hard to implement is for instance atomicity: if a process crashes while writing to the disk, either the write should succeed completely or leave no trace. In particular, the crashing process should never corrupt the disk itself.

3.3 Modelling Processes

We can also model processes. There is a wide range of choices in how to model a process in a distributed system and how they communicate. They can share memory or they can communicate solely via message passing. We will make a model based solely on message passing. We will also assume purely asynchronous programming: all inputs to a process are given as messages and they are queued on the receiving process in such a way that the "calling" process can proceed its execution immediately. In particular, the caller never waits for an output.



Protocol Ports versus Special Ports The ports of the process are divided into the protocol ports and the special ports. Protocol ports correspond to the ports the process has in "real life". The special ports are used only for modeling purpose. They are for instance used to model when a process is scheduled, when it crashes, and so on. We draw special ports with opened arrow heads. The protocol parts are in turn divided into two types of ports, the user interface ports (user ports) and the network interface ports (net ports). The user ports are those that the user of the system would call when it want to use the system. The net ports are the ones that the process calls when it needs to talk to other processes which are part of implementing the system.

Handling Input For each protocol port P the process has a queue $P.Q$. Initially the queue is empty. When the process receives a message m on a protocol port P it adds the message to $P.Q$. Then it returns DONE on P. Note, in particular, that the process performs no actions when activated on a protocol ports. In merely stores the received message for later use.

Activate The process has a special port called activate. When it receives any value on activate it can do one of the following any number of times:

- Read from a message queue $P.Q$.
- Change its internal state in any other ways.
- Send messages on some ports.

The activation ends when it returns any value on activate.

Figure 3.4 A Process.

Such pure asynchronous programming in general makes it easier to develop asynchronous systems and will in most cases be needed anyway. So we make it the default case here for simplicity.

A process will receive its input asynchronously as follows: when a message m is received on some port P, the message m is added to a queue Q. Then the process returns and does nothing more. There is a separate "thread" associated to the process which is the one that does the actual work. It proceeds in steps known

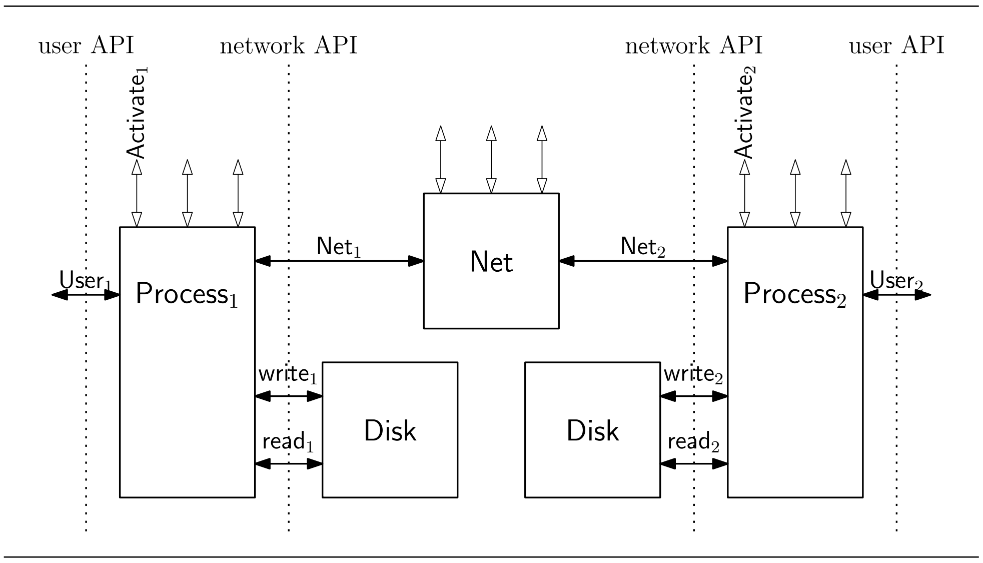


Figure 3.5 A Protocol with two processes, each with a disk. They use the network resource Net for communication. Note that the processes in a protocol have two types of protocol ports, the user ports where they take inputs for the protocol or middleware layer they try to implement and net ports on which they send messages to the network resources. To avoid confusing in later chapters it is important to have a clear distinction between these. We will call these ports the user API and tyhe network API of the protocol.

as activations. In each activation it is allowed to do some small amount of work and then the activation stops again. To model that processes by default might not progress at the same speed, we introduce a special port on which the process is told when to do the activations. Think of some evil daemon being connected to these ports. This means that in a larger system the daemon could try to activate the processes in a particular nasty order to break the system. This is an example of conservative modeling: we cannot by default assume that all processes proceed at the same speed, so to be on the safe side we assume that the order in which they take steps could be the worst possible one.

3.3.1 Specifying a Process

Formally a process is given by a transition function. But that is a very cumbersome formalism. Instead we will specify a process by a list of activation rules written in prose or pseudo-code. Here are some example of activation rules:

- On input $(\text{INCREMENT}, a)$ on port I, let $x \leftarrow x + a$ and send x on port P.
- On input (PING) on port P return on P.

- On input (CONDITIONAL-INCREMENT, a, b) on any port, if $x \leq b$, let $x \leftarrow x + a$ and send x on port P.

In general a rule has a trigger part of the form

On input (NAME, $v_1, v_2, \ldots$) on port P, where Cond.

In addition to the trigger part it has an action part A. This is the code that will be run if the rule is triggered. The action part is allowed to read the incoming queues, read local state, modify local state and send messages on outgoing port. It is not allowed to send anything on the activation port of the IA. Putting this parts together, an activation rule is of the form.

On input (NAME, $v_1, v_2, \ldots$) on port P, where Cond, A.

The activation rule is triggerable if in the queue P.Q there is a message m of the form (NAME, $v_1, v_2, \ldots$) and at the same time the condition Cond is true. The condition might depend on the message and the internal state of the process. When a process is activated on port activate it will go through all its activation rules from top to bottom. When it finds one that is triggerable, it triggers that rule.

If an activation rule is triggered, then the matching message m is removed from the queue. If the rule is triggered, then the algorithm A is run. This algorithm is allowed to run as a normal activation of a process with the restriction that it is not allowed to remove other messages from the message queues and is not allowed to send a message on the activation port. After running A, the process returns on activate.

If there is no activation rule which is triggerable, then it does nothing and immediately returns on activate. If there is no activation rule which is triggerable, then we say that the process is idle. Note that if a process is idle, then an activation cannot change that, as an activation in an idle state does not change the state of the process. Only the arrival of a new message might change the idle status of a process: the message might make some rules triggerable.

3.3.2 Some Possible Special Ports

In addition to activate a process might have other special ports. These special ports are typically used to model faults. Here are some examples of such ports.

crash On any input on the crash port, delete the process' set of activation rules. This means that the machine no longer does anything when activated.

takeover On input AR on takeover the activation rules of the process are replaced by the activation rules in AR. If the process models a machine, then this corresponds to a hacker completely taking over the machine and installing its own code on the machine.

leak On input L on leak, let σ denote the internal state of the process, including all the queues of incoming messages. Compute $y = L(\sigma)$. Then return y

on leak. This corresponds to the process leaking some information to its environment. If the leakage function L is the identity, then it corresponds to a hacker breaking into the machine and seeing all the data on the machine. In some case more limited leakage makes sense too. We later in the book discuss security problems arising from machines leaking their power consumption or exact running time of algorithms working on secret data.

Not all processes need have all these ports. It depends on the flavour of security we want to prove. We will always be explicit about these ports if they are there. So by default, if we do not say anything, a process only has the activate special port.

*On Leaky Crashes**

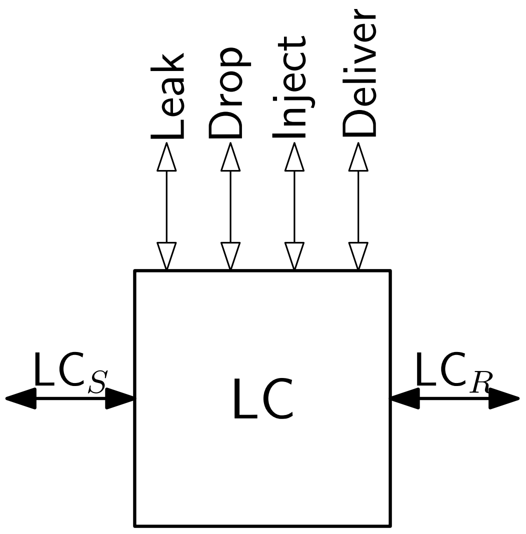
A note about how we handle crashes. The impetus to crash is given as an input from the outside of the machine. This implies that the time of a crash cannot depend on secrets inside the machine. We could have chosen another model, where some port is used to install a small piece of code which is watching the execution of the machine and then crashes it at a time depending on internal values on the machine. This better models crashes in real life: typically machines crash because of software bugs which occur at particular places in the code. Modeling state-dependent crashes would, however, make things very complicated when cryptography is used, since a part of the machine could be a secret key (we discuss secret keys in detail in Chapter 5 and following chapters). In this case an intruder could make the timing of the crash depend on the key, thus leaking secrets to the world via the crash. As an example, say that the intruder instructs the machine to crash after x milliseconds, where x is a 20-bit number containing the first 20 bits of the secret key. It is very hard to argue security when such leakage occurs. Such subtleties can be handled using leakage resilient cryptography, but this would go beyond the scope of this book.

3.4 Modelling Network Resources

When we model a protocol the parties need some means of communication. This will just be another IA modelling the means of communication we give to the protocol. We call such an IA a network resource. When we model a basic real world protocol, the network resources will typically be some model of the internet, and we call the network resources channels. But sometime we also want to assume that more complicated infrastructure is available to the part. This could for instance be a public-key infrastructure or a key distribution center.[2] In that case we use the generic name of network resources.

When we model network resources we do not have additional structure, except we make the same distinction between protocol ports (the ports modelling real-life input and output) and the special ports (used to model properties of the

[2] We discuss these notions later in the book.

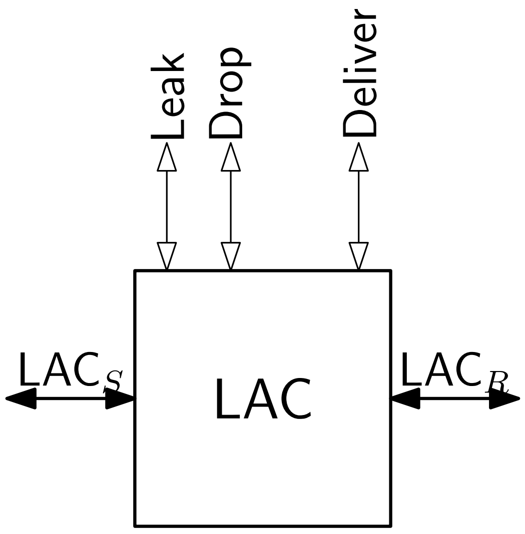


Ports The channel connects two parties with names S and R. For each party $P \in \{S, R\}$ it has a user port called LC_P. It has a special port called Leak, which is used to model that the channel does not hide what is sent on it from its environment. It has a special port Drop, which is used to model that it can drop messages. It has a special port INJECT, which is used to model that anyone can inject messages on the channel: it is not authenticated. Finally, it has a special port Deliver which is used to model that it is its environment which controls when messages are delivered.
Init It keeps a set InTransit which is initially empty.
Send On input (R, m) on LC_S, it outputs (S, R, m) on Leak, adds (S, R, m) to InTransit and then it returns on LC_S. Similarly for LC_R.
Drop On input (P, Q, m) on Drop it removes (P, Q, m) from InTransit and returns on Drop.
Inject On input (P, Q, m) on INJECT it adds (P, Q, m) to InTransit and returns on INJECT.
Deliver On input (P, Q, m) on Deliver where $(P, Q, m) \in$ InTransit, it removes (P, Q, m) from InTransit, outputs (P, m) on LC_Q and returns on Deliver.

Figure 3.6 Lossy Channel

network resource). On a network resource all the protocol ports are also called user ports and they are also called the user API. Note that this in particular means that when a process is connected to a network resource, then the network API of the process is connected to the user API of the network resource.

As an example, the IA in Fig. 3.6 models a lossy channel (LC), where the environment can see which messages are sent, it might drop a message and it might inject a message. Think of sending a message using UDP. The servers through which the message is sent can see it or drop it, and they might decide to forward a completely different message.



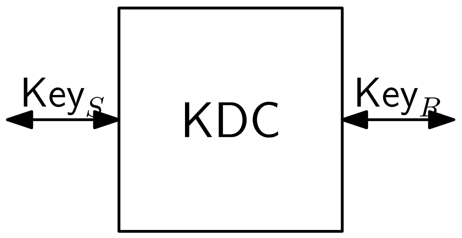
Ports The channel connects two parties with names S and R. For each party P it has a user port called LAC_P. It has a special ports Leak, Drop, and Deliver.
Init It keeps a set InTransit which is initially empty.
Send On input (R, m) on LAC_S, it outputs (S, R, m) on Leak, adds (S, R, m) to InTransit and then it returns on LAC_S. Similarly for LAC_R.
Drop On input (P, Q, m) on Drop it removes (P, Q, m) from InTransit and returns on Drop.
Deliver On input (P, Q, m) on Deliver where $(P, Q, m) \in$ InTransit, it removes (P, Q, m) from InTransit, outputs (P, m) on LAC_Q and returns on Deliver.

Figure 3.7 Lossy Authenticated Channel

As another example, the IA in Fig. 3.7 models a lossy authenticated channel (LAC), where the environment can see which messages are sent, and then might decide to drop a message. Think of sending a message using UDP but with some authenticity mechanism in place, for instance digital signatures (see Chapter 6). The servers through which the message is sent can see it or drop it, but they can't change it.

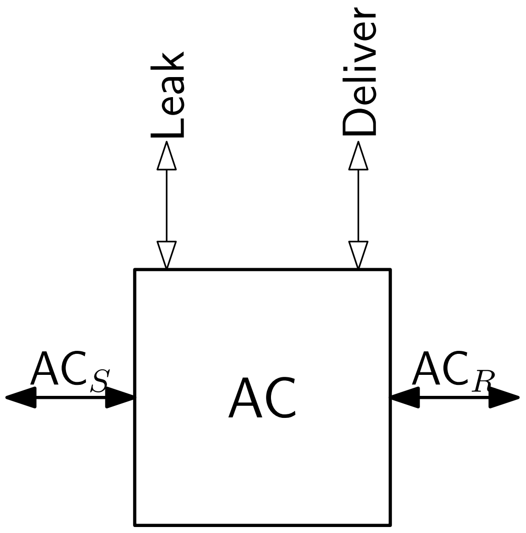
If we wanted to implement LAC from LC we would for instance need a symmetric key for a message authentication code (MAC). To be able to use a MAC the two parties will have to share a secret key, as we will discuss in great detail in Chapter 6. One way to put such a key in place is a so-called key distribution center which samples the key and somehow securely hands it to both parties. In Fig. 3.8 we model a very simple key distribution center for just two parties. It will generate a random key and securely send it to each of the parties. Importantly, this is done in a way such that the environment does not see the key K. We model this by not leaking K.

As our very last example, and mostly because we need it later, consider the



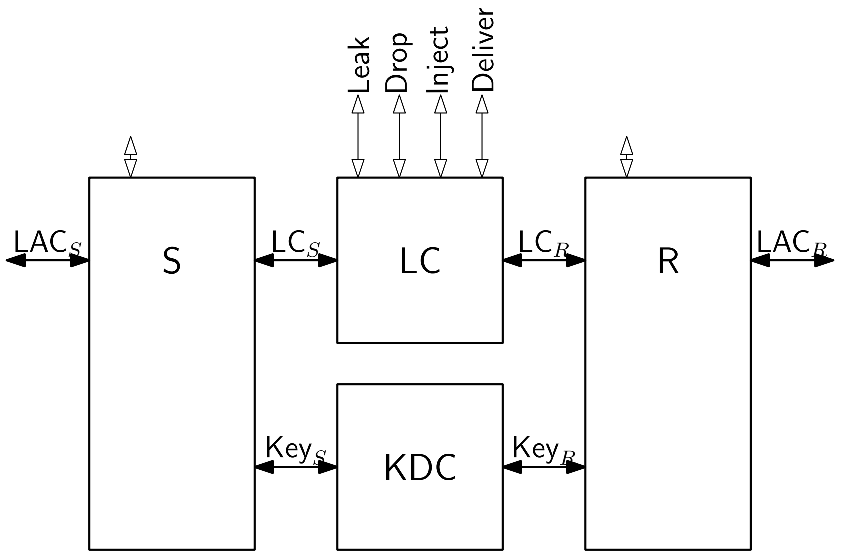
Ports The channel connects two parties with names S and R. For each party $P \in \{S, R\}$ it has a user port Key_P.
Generate Key It is parametrised by a key generation algorithm Gen. When activated on any port for the first time it samples a key $K \leftarrow \mathsf{Gen}$ and stores K.
Deliver On any input on Key_S, output K on Key_S. On any input on Key_R, output K on Key_R.

Figure 3.8 KDC



Ports The channel connects two parties with names S and R. For each party P it has a port called AC_P. It has a special ports Leak and $\mathsf{Deliver}$.
Init It keeps a set $\mathsf{InTransit}$ which is initially empty.
Send On input (R, m) on AC_S, it outputs (S, R, m) on Leak, adds (S, R, m) to $\mathsf{InTransit}$ and then it returns on AC_S. Similarly for AC_R.
Deliver On input (P, Q, m) on $\mathsf{Deliver}$ where $(P, Q, m) \in \mathsf{InTransit}$, it removes (P, Q, m) from $\mathsf{InTransit}$, outputs (P, m) on AC_Q and returns on $\mathsf{Deliver}$.

Figure 3.9 Authenticated Channel



Both parties $P \in \{R, S\}$ run this code:

- If no K is stored, call KDC.Key if this was not already done.
- On output K on KDC.Key, store K.
- On (Q, m) on LC2LAC.LAC$_P$, add (Q, m) to InTransit.
- If K is stored and there is some $(Q, m) \in$ InTransit then compute $c = \mathrm{MAC}_K(P, Q, m)$ and input (m, c) on LC.LC$_P$.
- On output $(Q, (m, c))$ on LC.LC$_P$ where K is stored check that $c = \mathrm{MAC}_K(Q, P, m)$. If so, output (Q, m) on LC2LAC.LAC$_P$.

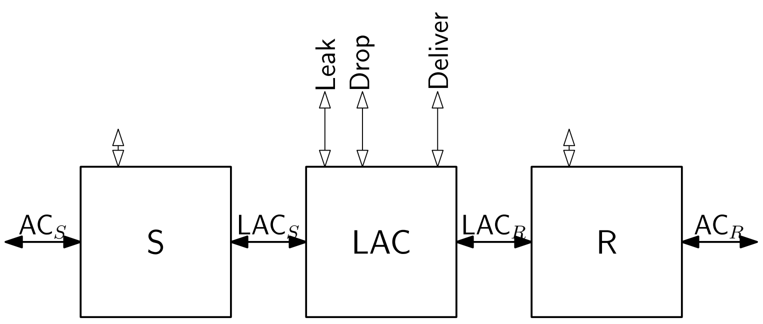
Figure 3.10 The LC2LAC protocol. It uses two network resources, LC and KDC. It has two processes, S and R. They are connected to the network resources as shown in the figure. The unnamed special port that each process has denotes the activate port, which we for brevity will stop showing explicitly. Party P in addition has a port called LC2LAC.LAC$_P$ (in the figure just called LAC$_P$. The special ports on the parties are there simply as placeholders to remind us that the parties too could have special ports to for instance model corruption.)

authenticated channel (AC) in Fig. 3.9. It works as a LAC, but there is no special command to drop messages. So if a message is sent, then it is eventually delivered.

3.5 Modelling Protocols

We are then ready to model protocols. Protocols will consist of processes, network resources and disks connected to each other in some way, see Fig. 3.5 for a generic example. Note that disks are considered special cases of network resources, which just happen to connect to only one process. As another example, consider the protocol LC2LAC in Fig. 3.10. It tries to turn a LC into a LAC. For this it uses a message authentication code (MAC). We discuss MACs later in (see Chapter 6). What is important is that for a fixed key, there is a function MAC_K which maps a string x to a code c, and you can compute $c = \mathrm{MAC}_K(x)$ only if you know K. So when R receives c such that $c = \mathrm{MAC}_K(S, R, m)$, then R knows that it was S who sent the message.

As another example, consider the LAC2AC protocol in Fig. 3.11. It tries to turn a LAC into an AC. We have given each party a disk. They don't use them for now, but they come in handy later.



Sender Code

- On INIT, create an empty set InTransit.
- On (R, m) on Send$_S$, add (R, m) to InTransit
- If InTransit is not empty retrieve the oldest (R, m) from InTransit and send m to R using LAC.
- If (ACK, m) arrives from R on LAC, then remove (R, m) from InTransit.

Receiver Code

- On INIT, create an empty set Delivered.
- On m from S via LAC, where $(S, m) \notin$ Delivered, proceed as follows:
  1. Output (S, m) on Receive$_S$.
  2. Add (S, m) to Delivered.
  3. Send (ACK, m) to S using LAC.
- On m from S via LAC, where $(S, m) \in$ Delivered, send (ACK, m) to S using LAC.

Figure 3.11 The LAC2AC protocol

The intuition behind the LAC2AC protocol is that we just keep sending the message until it gets through. If we try an infinitely number of times, then any reasonable channel should allow the message to get through at some point. When this happens, the message will be delivered at R's end. At the same time an acknowledgement is sent back. If it is lost, then S keeps sending, so eventually the message gets through again and a new acknowledgement is sent. And so on. So eventually the acknowledgement must get through. At that time S stops sending. After some time all messages are then delivered or dropped and the protocol becomes silent. The only trace that will be left is that (S, m) is stored at R in Delivered. We will discuss later whether we can clean that up.

3.5.1 Ideal Functionalities

As should be clear from the above, network resources are used for two purposes in our model. They can be used as the network resource of a protocol, like when we use LAC in LAC2AC to implement AC. They can also be used to *specify* what a concrete protocol is supposed to do. We for instance want LAC to behave like AC. When a network resource is used to specify what a system is supposed to do we sometimes call it an ideal functionality. Notice that when we use AC as an ideal functionality to specify what LAC2AC is supposed to do, then there is no AC present in LAC2AC. We just compare LAC2AC to AC. On the other hand, when we use a network resource as a sub-protocol, like when we use LAC in LAC2AC, then there is in fact a copy of LAC present in LAC2AC.

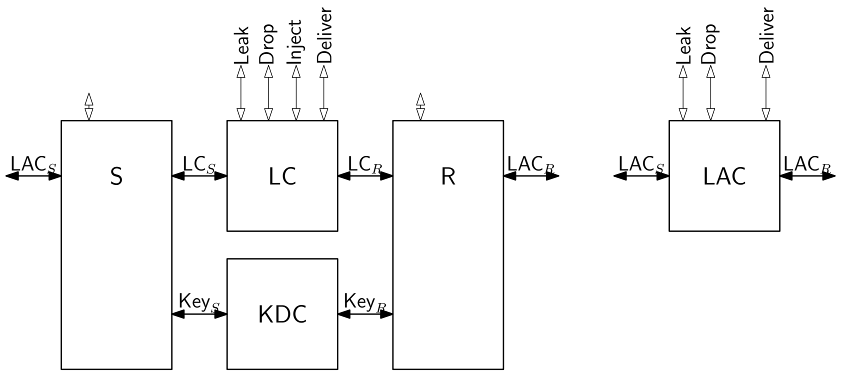
3.6 Specifying Safety Properties

The above text should give a very rough idea how of we formalise what a protocol does. We now discuss how one would specify what it is *supposed* to do. Recall that we typically want to specify two different kinds of properties, safety properties and liveness properties. Safety properties basically specify what is the correct relation between the inputs and outputs of the system. A safety property is only broken if the system gives a wrong output. Liveness properties specify that the system must make progress, for instance that it does not have deadlocks. A liveness property is only broken if the system ends up in a state where there is still work to do but the state of the system prevents it from ever making progress again. We start by looking at how to specify safety properties.

What is the intended behaviour of LC2LAC? Well, we stated that we wanted it to implement a LAC. And, we already specified what we think a LAC is. We did that in Fig. 3.7. So there is no reason to complicate things. We simply state that the protocol should behave as the network resource we wanted it to be. Here is the first attempt at a definition:

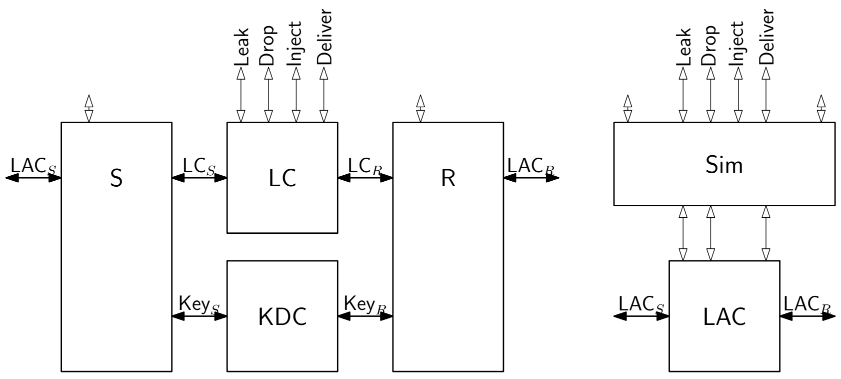
— *The protocol* LC2LAC *should behave as the network resource* LAC. —

We then just have to make *"bahave as"* more precise and we are done. When we address the safety of the system, what we want to say is that the protocol LC2LAC will not give any outputs that LAC could not have given. For instance, if LC2LAC would deliver to R a message m' that S did not send, then safety is broken as LAC would never deliver to R a message m' that S did not send. We now unfold this informal definition a bit. Let us first take a look at the protocol and the network resource it is supposed to implement next to each other:



When we say that the left system behaves as the right system, we will only be concerned about the protocol ports LAC_S and LAC_R. After all, these are the ports modeling the actual IO behaviour of the protocol. The special ports just model all the open-ended parts of what the network resource might do. Yet, to know what the system actually outputs in response to inputs on the protocol ports, we also need to know what are the inputs on the special ports. We cannot require that the inputs on the special ports are the same too, as the special ports of the two systems might have different roles. After a moment of thought the right definition is to require that for each sequence of inputs to the protocol ports and special ports of LC2LAC there exists a sequence of inputs to the protocol ports and special ports of LAC that makes it give the same outputs as LC2LAC. Furthermore, the input to the protocol ports must be the same in the two sequences, but the inputs to the special ports might differ.

There is an elegant way to turn that into a very precise definition. One requires that there exists a so-called simulator Sim that can be connected to the special ports of LAC such that the protocol and LAC+Sim has exactly the same behaviour. Note that this means that the simulator has all the same special ports as LAC and LC. It will connect to the special port of LAC and hence the open special ports of Sim in Sim + LAC will be exactly those of LC. That is why we call it a simulator: it pretends to be an LC. The job of the simulator is to make sure that Sim + LAC behaves like S + LC + KDC + R, i.e., on any sequence of inputs they produce the same outputs.



Notice that after adding Sim the two systems have exactly the same open ports so it now makes sense to require that they have the same input-output behaviour.

Definition 3.1 (security) We say that a protocol π securely implements a network resource C if there exists an interactive agent Sim that connects only to the special ports of C and such that π and $C + \mathsf{Sim}$ have the same input output behaviour. We write $\pi \sqsupseteq C$.

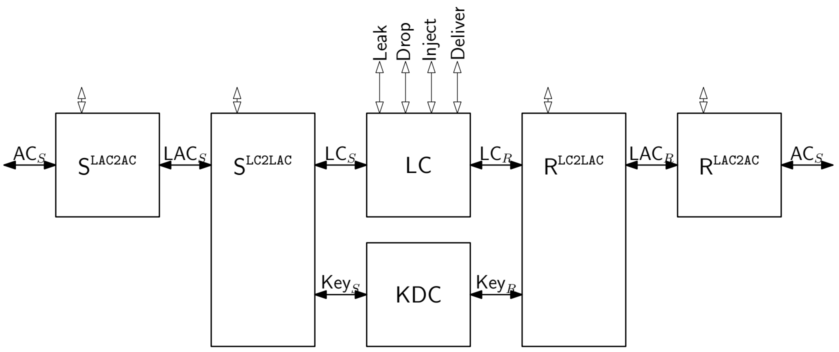


Figure 3.12 The composition LC2AC of LC2LAC and LAC2AC running with network resource LC. In the picture we annotate S and R with the protocol from which the IA was taken to disambiguate the IA called S and R in the LC2LAC and LAC2AC protocols.

3.7 Composition

An important property of our model of security is so-called composition. It is easier to look at an example of what it means. Above we saw a protocol LAC2AC securely implementing AC from LAC. We also saw a protocol LC2LAC which implements LAC from LC. It seems reasonable that if we combine them then we can obtain a protocol LC2AC which securely implements AC from LC. This is indeed the case. If we replace the LAC in LAC2AC by the protocol LC2LAC we get a protocol using the network resource LC and it is indeed the case that it implements AC. The composed protocol is depicted in Fig. 3.12.

It seems reasonable that LC2AC securely implements AC. But can we be sure? Maybe there is some subtlety in the protocol which makes it secure when it uses LAC but insecure when it uses LC2LAC. Do we have to reprove security of the composed protocol to be on the safe side? It turns out that we do not have to reprove security. It can be proven once and for all that composition maintains security. If a protocol security implements a network resource then it can be plugged into any secure protocol using that resource, and security is maintained.

We will not prove here that our framework has secure composition. For the formally inclined reader, the topic is explored in more detail in Section 3.10. The topic of composition of secure protocols is explored in great detail in for example the work on the Universal Composition framework[3] introduced by Ran Canetti and the work on the Constructive Cryptography framework introduced by Ueli Maurer[6]. Our exposition here uses elements from both of the frameworks.

3.8 Specifying Liveness Properties

Notice that the above definition does not capture liveness properties. If LC2LAC had a deadlock that at some point prevented it from ever delivering messages again, then the simulator could simply stop instructing LAC to deliver messages. The systems would have the same input-output behaviour. To have liveness we want that LC2LAC is guaranteed to always make progress: if a message is sent then it is eventually delivered. It would, however, be unfair to require LC2LAC to have this property unconditionally. It could be that LC drops all messages. In that case clearly LC2LAC cannot deliver its own messages. So what we really want to require is that *if* the network resources used by the protocol (here KDC and LC) are live, then the protocol (here LC2LAC) must also be live. So in some sense we want to require that the protocol is liveness preserving. For the case of LC2LAC it is easy to see that this is the case even without a formal definition.

It is perhaps a bit harder with LAC2AC: is LAC2AC liveness preserving? To answer this we first need to define what it means to be live for LAC and AC. There is no easy way to pick these definitions correctly, so the following discussion is going to be much less formal than what we did for safety properties:

- We say that LAC is live if it guarantees the following: any message that is sent "sufficiently many times" is eventually delivered.
- We say that AC is live if it guarantees the following: any message that is sent is eventually delivered.

The difference between the definitions take into account the properties of the two channels. Remember that the special ports of LAC allow the environment to drop messages. Therefore it does not make sense to ask that all messages are eventually delivered. On the other hand the channel AC does not have a special drop port, therefore messages will stay in the queue of the channel forever, waiting to be eventually delivered.

Finally note that defining what eventually means in a precise way is very subtle task. However, it should be clear when looking at the LAC2AC protocol that if the underlying LAC channel guarantees to deliver any message that is sent "sufficiently many times", then LAC2AC will also "eventually" deliver any message sent via the protocol.

3.9 Handling Crashes

We now focus on handling crashes in the LAC2AC protocol. We will assume that both S and R can do crash-recovery in the crash-silent model (remember, this

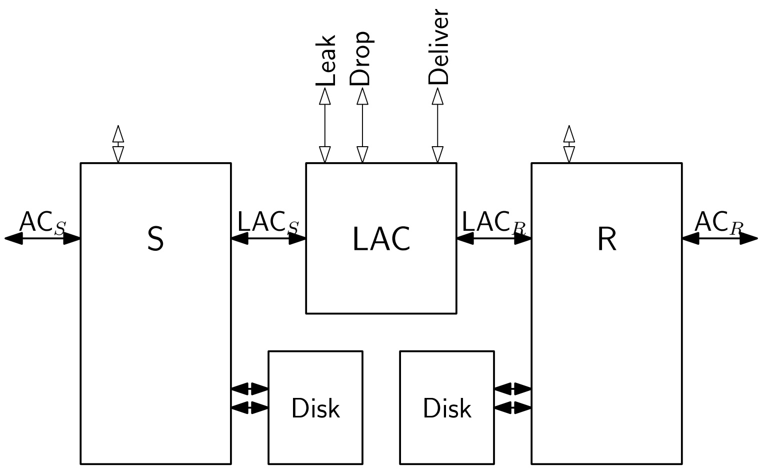


Figure 3.13 The architecture with a lossy channel and two disks

is the model in which a process crashes without sending any message to other processes). When they crash they come back to live in the initial state. Their disks survive the crash. We would like to make a system which ensures that if (R, m) is sent once at S, then (S, m) is delivered exactly once at R. A naïve solution would be the following: run as in LAC2AC, but keep the set of delivered messages (called Delivered in the LAC2AC protocol) on disk. Note, however, that the following could happen: S outputs (S, m). Then it crashes right before it gets to add (S, m) to Delivered. When it comes back to life, it will eventually receive (S, m) again and deliver it another time. One could then try to first add (S, m) to Delivered and only afterwards output it. But then clearly we get the dual problem: R could crash right after adding (S, m) to Delivered and before outputting it. When it comes back to life it will find (S, m) in Delivered and will therefore never deliver (S, m). The first implementation guarantees that the message is delivered at least once, but might make duplicates. The second implementation guarantees that a message is delivered at most once, but might never deliver it. What we really want is a channel which delivers the message exactly once. It turns out that this is impossible, at least with the current system architecture.

We will use this as a chance to prove our first impossibility result. Consider any protocol π for an architecture as in Fig. **??**. We will prove that π cannot guarantee to deliver m exactly once, no matter how π works. The argument goes as follows. Send (R, m) on π and then activate all processes and channels sufficiently many times. Since π delivers m exactly once, there will be a point in time where R is about to output (S, m). Call this point p. We can crash R just before p or just after and compare what will happen. If we crash it just before p, then it does not output (S, m). So when it comes back to life, it will eventually deliver (S, m) at some future point q—recall that it delivers the message m once, so it must deliver

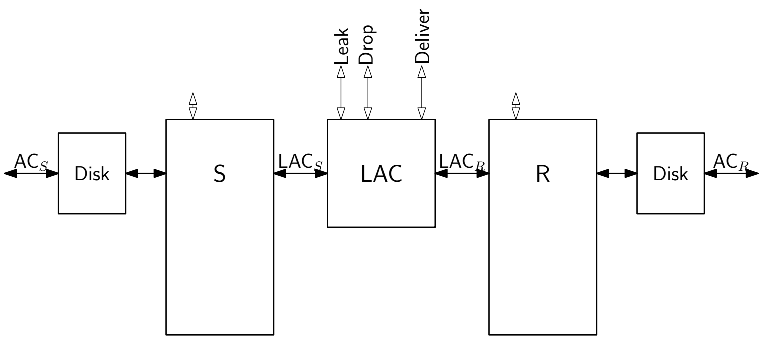


Figure 3.14 The architecture with a lossy channel and two disks used for input and output. If a party crashed the contents on the disk survives, so one cannot forget have have gotten an input or having given an output.

it when it comes back to life, as it was not delivered yet. However, if we crash R right after delivering (S, m) and we let it recover, then it will be in exactly the same state as when we crashed it just before p. In fact, the entire system is in the same state. Hence if we run the system, from this state, then it will as before deliver (S, m) at q. But then it delivered (S, m) twice. This shows that you can never get *exactly once*.

Note that the above argument assumes that the system is crashed exactly at some bad point. Recall that our model of crash is that the environment picks the point where we crash. So maybe we could try to hide the time of the bad point from the environment. If we always place the bad point at the same point in the execution, or at some point which can be computed deterministically by the environment, it will not work. So we need to try to place the bad point at some random point.

Exercise 3.1 (probabilistic solution 1) Try to build a system which never delivers a message more than once and which with probability 1/2 will deliver it at least once. Can you get a lower probability?

Exercise 3.2 (probabilistic solution 2) Try to build a system which always delivers a message at least once and which with probability 1/2 will not deliver it more than once. Can you get a lower probability?

A popular solution to the problem that you have to pick at-least-once or at-most-one is to pick at-least-once and then make sure that the outer system using the channel can tolerate that a message is delivered more than once. One can for instance use sequence numbers.

Another popular solution is to define that a message m is delivered when R added (S, m) to its disk. Similarly it is said to be sent when (R, m) was added to the disk of S.

Exercise 3.3 (more crashes) Make a system where outputs are defined to be given via the disks and which can tolerate crash-recovery of both S and R and which implements *exactly-once*. In a bit more detail, there is a sender S and a receiver R. Each of them have a disk. The users of the channel can also write to their disk. A message m is said to have been sent when it is written to the disk of S by the user of S. It is said to have been delivered when it is written to the disk of R by R. There it will be picked up by the user of R. It turns out that one can get "exactly once" while using only one disk. There is no needed to use a disk at both the sending end and the receiving end.

- Show how to get exactly-once while only using the disk of R.
- Show how to get exactly-once while only using the disk of R.

In the above solutions you probably left something on disk even after the protocol was done. It turns out that you have to.

Exercise 3.4 (cleaning up) Show that in a protocol for the architecture in Fig. 3.14 it is not possible to get a protocol which has exactly once delivery and where the disks are completely empty again efter the protocol terminated.

3.10 Composition*

An important property of a model of security is so-called composition. It is easier to look at an example of what it means. Above we saw two protocols, LC2LAC and LAC2AC. The first one securely implements LAC from LC and is liveness preserving. The second one securely implements AC from LAC and is liveness preserving. It seems intuitive that if we combine the protocol, then we get a protocol which securely implements AC from LC. The composed protocol would look as in Fig. 3.12.

Can we be sure that the combined protocol securely implements AC? Can we be sure that it is liveness preserving?

Composability is an essential property of any definition of security. If composability holds in the model, then we can prove the properties of the combined protocol in a modular way. This allows us to given a simpler and modular analysis where each step focuses only on one small part of the system (for instance, turning a lossy channel into a non-lossy channel). Then, composability would guarantee that security of the individual steps gives security of the composed protocol. If a model does *not* satisfy composability, then each new protocol like LC2AC would have to be analysed from scratch.

Fortunately for us the model we gave above satisfies composability. Proving this in detail is out the scope of this book, but we would like to give the interested reader an idea why this is the case.

Below we list some results about how the model composes, with some intuitive explanations. A good way to remember the composition rules is basically that behavioural equivalence, $\equiv$, is like equality on integers and secure implementation,

$\sqsubseteq$, behaves as $\leq$ on integers. Another popular way to read $\mathsf{S}_1 \sqsubseteq \mathsf{S}_2$ is as "S_1 is at least as secure as S_2".

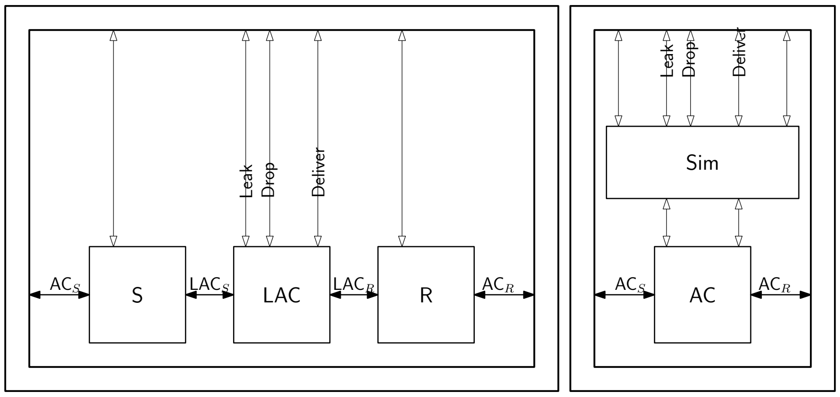


Figure 3.15 Comparing $\mathsf{LAC2AC} + \mathsf{LAC}$ and $\mathsf{LAC} + \mathsf{Sim}$ using an environment $\mathcal{Z}$. The environment is shown as the IA surrounding the other systems.

First we need to define behavioural equivalence with a bit of care. Recall that we say that two systems S_0 and S_1 behave the same if they have the same outputs given the same inputs. It turns out this is too strict. It is OK if there is a slight difference in the probability of outputting one particular value as long as this cannot be noticed by any plausible attacker. So, if we want to compare two system like $\mathsf{LAC2AC}+\mathsf{LAC}$ and $\mathsf{LAC}+\mathsf{Sim}$ we would like to say that it should hold for all plausible adversaries $\mathcal{Z}$ that it cannot guess whether it is in the setting $\mathcal{Z}+(\mathsf{LAC2AC}+\mathsf{LAC})$ or the setting $\mathcal{Z}+(\mathsf{LAC}+\mathsf{Sim})$. See Fig. 3.15 for an illustration.

We will sketch how to define this. First let $\mathsf{S}_0 = \mathsf{LAC2AC} + \mathsf{LAC}$ and $\mathsf{S}_1 = \mathsf{LAC} + \mathsf{Sim}$. Then sample a uniformly random bit $b \in \{0,1\}$. Then run $\mathsf{S}_b + \mathcal{Z}$. Assume that $\mathcal{Z}$ always outputs a guess $g \in \{0,1\}$ at the end. This is a guess at whether it played with S_0 or S_1. It can always make a random guess and hence win with probability $\frac{1}{2}$. So to do well it needs to be correct with probability *better* than $1/2$. How much it is better is called its advantage. We write it as

$$\textsc{Adv}_{\mathcal{Z}}(\mathsf{S}_0, \mathsf{S}_1) = \Pr[g = b] - 1/2 \ .$$

We say that they systems behave *exactly the same* if it holds for *all* $\mathcal{Z}$ that it guesses correct with probability at most $1/2$, i.e., $\textsc{Adv}_{\mathcal{Z}}(\mathsf{S}_0, \mathsf{S}_1) \leq 0$. We say that S_0 and S_1 are *behaviourally equivalent* if for all *plausible* $\mathcal{Z}$ it holds that $\mathcal{Z}$ guesses correctly with a probability *not too much higher than* $1/2$. To define "plausible" we could for instance pick some number of computational steps that we think no adversary can perform in practice, like 2^{100}, and say that a plausible adversary is one that performes less than 2^{100} computational steps. To define "not too much higher than" we need to pick some probability ϵ that we consider negligible, for instance $\epsilon = 2^{-80}$. Then we say that the adversary should not be able to guess correctly with probability better than $1/2 + \epsilon$. When we take an IA like $\mathcal{Z}$ which connects to all open ports of S_0 (and S_1) we call it an environment of S_0 (and S_1). This gives us a definition as follows.

Definition 3.2 (behavioural equivalence (informal)) Let S_0 and S_1 be systems with the same open protocol ports and the same open special ports. We say that S_0 and S_2 are behaviourally equivalent if for all plausible environments $\mathcal{Z}$ there is a negligible ϵ such that $\text{Adv}_{\mathcal{Z}}(\mathsf{S}_0, \mathsf{S}_1) \leq \epsilon$.

The notion of secure implementation can be extended to any two systems, not just as a relation between protocol and network resources.

Definition 3.3 (general secure implementation) Let S_1 and S_2 be systems with the same protocol ports and possibly different special ports. We say that S_1 securely implements S_2, written as $\mathsf{S}_1 \sqsubseteq \mathsf{S}_2$, if there exists an efficient simulator Sim such that $\mathsf{S}_1 \equiv \mathsf{S}_2 + \mathsf{Sim}$.

We already argued why this definition makes sense. The reason why we require Sim to be efficient is that otherwise composition does not work. The definition of "efficient" we leave out, but it should hold that any "efficient" IA is also "plausible". Note how the definition parallels the definition of $\leq$ on natural numbers: We say that $s_1 \leq s_2$ if there exists non-negative s such that $s_1 = s_2 + s$.

Assumption 3.4 (transitivity of $\equiv$) *Let S_1, S_2, and S_3 be systems such that $\mathsf{S}_1 \equiv \mathsf{S}_2$ and $\mathsf{S}_2 \equiv \mathsf{S}_3$. Then $\mathsf{S}_1 \equiv \mathsf{S}_3$.*

It would be strange is a notion of "behaving the same" was not transitive, so the above ought to be a triviality. But note that we did not define $\equiv$ very precisely yet. This can be done such that it has the above property. There are some technicalities to be handled carefully, but it is not too complicated.

Note that the parallel statement for natural number is trivial: if $s_1 = s_2$ and $s_2 = s_3$, then $s_1 = s_3$.

Assumption 3.5 (extension of $\equiv$ (informal)) *Let S_1 and S_2 be systems for which $\mathsf{S}_1 \equiv \mathsf{S}_2$. Let S be any efficient system. Then $\mathsf{S}_1 + \mathsf{S} \equiv \mathsf{S}_2 + \mathsf{S}$.*

This again ought to be the case if $\equiv$ has been defined correctly. If S_1 and S_2 give the same outputs on the same inputs, so clearly does $\mathsf{S}_1 + \mathsf{S}$ and $\mathsf{S}_2 + \mathsf{S}$. The S_i parts of the systems behave the same, and the S part *is* the same. So $\mathsf{S}_1 + \mathsf{S}$ and $\mathsf{S}_2 + \mathsf{S}$ will behave the same. Again, one will craft the definition of $\equiv$ and "efficient" to have this property.

The parallel statement for natural numbers would be: if $s_1 = s_2$ then $s_1 + s = s_2 + s$.

Lemma 3.6 (extension of $\sqsubseteq$) *Let S_1 and S_2 be systems for which $\mathsf{S}_1 \sqsubseteq \mathsf{S}_2$. Let S be an efficient system only connecting to the protocol ports of S_1 and S_2. Then $\mathsf{S} + \mathsf{S}_1 \sqsubseteq \mathsf{S} + \mathsf{S}_2$.*

PROOF The reason is that $\mathsf{S}_1 \sqsubseteq \mathsf{S}_2$ implies that there exists Sim such that $\mathsf{S}_1 \equiv \mathsf{S}_2 + \mathsf{Sim}$. Since Sim connects only to special ports of S_2 and S connects only to protocol ports, it makes sense to consider the system $(\mathsf{S}_2 + \mathsf{Sim}) + \mathsf{S}$. From $\mathsf{S}_1 \equiv \mathsf{S}_2 + \mathsf{Sim}$ we get that $\mathsf{S}_1 + \mathsf{S} \equiv (\mathsf{S}_2 + \mathsf{Sim}) + \mathsf{S}$. Clearly $(\mathsf{S}_2 + \mathsf{S}) + \mathsf{Sim} = (\mathsf{S}_2 + \mathsf{Sim}) + \mathsf{S}$. Putting these together we get that $\mathsf{S}_1 + \mathsf{S} \equiv (\mathsf{S}_2 + \mathsf{S}) + \mathsf{Sim}$. This shows that $\mathsf{S}_1 + \mathsf{S} \sqsubseteq \mathsf{S}_2 + \mathsf{S}$, with Sim being the required simulator. □

The parallel statement for natural numbers would be: if $s_1 \leq s_2$ then $s_1 + s \leq s_2 + s$.

Lemma 3.7 (transitivity of $\sqsubseteq$) *Let S_1, S_2, S_3 be systems for which $\mathsf{S}_1 \sqsubseteq \mathsf{S}_2$ and $\mathsf{S}_2 \sqsubseteq \mathsf{S}_3$. Then $\mathsf{S}_1 \sqsubseteq \mathsf{S}_3$.*

PROOF From $\mathsf{S}_1 \sqsubseteq \mathsf{S}_2$ we get Sim_1 such that From $\mathsf{S}_1 \equiv \mathsf{S}_2 + \mathsf{Sim}_1$. From $\mathsf{S}_2 \sqsubseteq \mathsf{S}_3$ we get Sim_2 such that From $\mathsf{S}_2 \equiv \mathsf{S}_3 + \mathsf{Sim}_2$. By extension of $\mathsf{S}_2 \equiv \mathsf{S}_3 + \mathsf{Sim}_2$ with Sim_1 we get that $\mathsf{S}_2 + \mathsf{Sim}_1 \equiv \mathsf{S}_3 + \mathsf{Sim}_2 + \mathsf{Sim}_1$. From transitivity of $\equiv$ and $\mathsf{S}_1 \equiv \mathsf{S}_2 + \mathsf{Sim}_1$ and $\mathsf{S}_2 + \mathsf{Sim}_1 \equiv \mathsf{S}_3 + \mathsf{Sim}_2 + \mathsf{Sim}_1$ we get that $\mathsf{S}_1 \equiv \mathsf{S}_3 + \mathsf{Sim}_2 + \mathsf{Sim}_1$. If we let $\mathsf{Sim} = \mathsf{Sim}_2 + \mathsf{Sim}_1$, this gives us that $\mathsf{S}_1 \equiv \mathsf{S}_3 + \mathsf{Sim}$, which shows that $\mathsf{S}_1 \sqsubseteq \mathsf{S}_3$. □

The parallel statement for natural numbers would be: if $s_1 \leq s_2$ and $s_2 \leq s_3$, then $s_1 \leq s_3$.

Theorem 3.8 (general universal composition of $\sqsubseteq$) *Let S_1, S_2, S_3, and S_4 be efficient systems for which $\mathsf{S}_1 \sqsubseteq \mathsf{S}_2$ and $\mathsf{S}_3 + \mathsf{S}_2 \sqsubseteq \mathsf{S}_4$. Then $\mathsf{S}_3 + \mathsf{S}_1 \sqsubseteq \mathsf{S}_4$.*

PROOF By extension of $\mathsf{S}_1 \sqsubseteq \mathsf{S}_2$ with S_3 we get that $\mathsf{S}_1 + \mathsf{S}_3 \sqsubseteq \mathsf{S}_2 + \mathsf{S}_3$. Then use $\mathsf{S}_3 + \mathsf{S}_2 = \mathsf{S}_2 + \mathsf{S}_3$ and $\mathsf{S}_3 + \mathsf{S}_2 \sqsubseteq \mathsf{S}_4$ and transitivity of $\sqsubseteq$ and you get that $\mathsf{S}_1 + \mathsf{S}_3 \sqsubseteq \mathsf{S}_4$, as desired. □

From the above we have get the following result for composability of secure protocols.

Corollary 3.9 (universal composition of $\sqsubseteq$) *Let $\mathcal{N}_1$, $\mathcal{N}_2$, and $\mathcal{N}_3$ be network resources. Let Π_1 and Π_2 be protocols. If , and S_4 be efficient systems and let $\mathcal{N}_1$ for which $\Pi_1 + \mathcal{N}_1 \sqsubseteq \mathcal{N}_2$ and $\Pi_2 + \mathcal{N}_2 \sqsubseteq \mathcal{N}_3$. Then $\Pi_1 + \mathsf{P}_2 + \mathcal{N}_2 \sqsubseteq \mathcal{N}_3$.*

PROOF Use general universal composition with $\mathsf{S}_1 = \Pi_1 + \mathcal{N}_1$, $\mathsf{S}_2 = \mathcal{N}_2$, $\mathsf{S}_3 = \Pi_2$, $\mathcal{N}_3 = \mathsf{S}_4$. □

The parallel statement for natural numbers would be: if $s_1 \leq s_2$ and $s_3 + s_2 \leq s_4$, then $s_3 + s_1 \leq s_4$.

3.11 On Defining Eventuality*

For the formally inclined reader, we want to add a few details about how to define "eventual".

We take LAC as an example. Consider LAC as sitting in the $\mathsf{LAC2AC}$ protocol. We would like to talk about running this system. For that we need someone or something to give it inputs. For this purpose, consider an adversary $\mathcal{Z}$ connected to all the open ports of the system. This is also sometimes called an environment. The environment has no other ports than those connected to LAC2AC, except that it has one open port which is its activation port. This is illustrated to the left in Fig. 3.15.

We can then run $\mathsf{LAC2AC} + \mathcal{Z}$ by activating $\mathcal{Z}$ and letting it play with LAC2AC as it wants until it terminates. So it is the environment that gives inputs on the protocol ports, receives outputs on the protocol ports and which controls all the special ports. The execution will proceed via a number of activations of LAC2AC. We want to force $\mathcal{Z}$ to eventually deliver messages sent on LAC. We can do this as follows: When the execution begins, $\mathcal{Z}$ commits to an integer $E > 0$, say by writing it down in some designated part of its state. This is now its eventuality bound. We can for instance use the bound E as follows: if the same message is sent on LAC at least E times, then $\mathcal{Z}$ does not drop it all E times. Furthermore, if $\mathcal{Z}$ activated the system LAC2AC at least E times while there was a message in transit in LAC2AC, it must deliver at least one of these messages. These two restrictions on $\mathcal{Z}$ forces it to sometimes deliver some message that was sent often enough. We call such environments E-restricted, and we write $\mathcal{Z}^E$. When we say that LAC eventually delivers a message, we then simply mean that we only consider executions that take place in E-restricted environment.

Note that we could make a similar definition for AC. Consider an environment $\mathcal{Z}'$ connected to all ports of AC. We say that $\mathcal{Z}'$ is E'-restricted if whenever it ran for E' activations while some message was in transit in AC, it will deliver at least one such message.

Recall then that we define security by requiring that there exists a simulator Sim such that $(\mathsf{S} + \mathsf{LAC} + \mathsf{R})$ and $\mathsf{AC} + \mathsf{Sim}$ have the same behaviour. This in particular means that $(\mathsf{S} + \mathsf{LAC} + \mathsf{R}) + \mathcal{Z}$ and $(\mathsf{AC} + \mathsf{Sim}) + \mathcal{Z}$ have the same behaviour for all $\mathcal{Z}$. This trivially implies that $(\mathsf{S} + \mathsf{LAC} + \mathsf{R}) + \mathcal{Z}^E$ and $(\mathsf{AC} + \mathsf{Sim}) + \mathcal{Z}^E$ have the same behaviour for all E-restricted $\mathcal{Z}^E$. Note that we can rewrite $(\mathsf{AC} + \mathsf{Sim}) + \mathcal{Z}^E$ as $\mathsf{AC} + (\mathsf{Sim} + \mathcal{Z}^E)$. In that case we can consider $\mathcal{Z}' = \mathsf{Sim} + \mathcal{Z}^E$ as an environment for AC.

Recall that we want to define that LAC eventually delivers messages when running in $(\mathsf{S} + \mathsf{LAC} + \mathsf{R}) + \mathcal{Z}^E$. But since this is a property that can be verified by only looking at the ports of the system and $(\mathsf{S} + \mathsf{LAC} + \mathsf{R}) + \mathcal{Z}^E$ and $(\mathsf{AC} + \mathsf{Sim}) + \mathcal{Z}^E$ have the same behaviour on the open ports, this is the same as requiring that AC eventually delivers messages $(\mathsf{AC} + \mathsf{Sim}) + \mathcal{Z}^E$, which is the same as saying that AC eventually delivers messages in $\mathsf{AC} + \mathcal{Z}'$. Recall then that we say that AC eventually delivers messages if $\mathcal{Z}'$ is E' restricted for some integer E'. This leads to the following simple definition: For all E there exists E' such that $\mathsf{Sim} + \mathcal{Z}$ is E'-restricted when $\mathcal{Z}$ is E-restricted. We call such a simulator liveness preserving.

To see that the definition makes sense, note that it essentially just says that of Sim is simulating an execution where LAC eventually delivers messages, then Sim must also eventually deliver messages. And since the simulation looks like the real protocol, we get that in an execution where LAC eventually delivers messages, the protocol LAC2AC also eventually delivers messages.

So, we can recap the definitions as follows: We call $\mathsf{S} + \mathsf{LAC} + \mathsf{R}$ a secure implementation of AC if there exists a simulator Sim such that $\mathsf{S} + \mathsf{LAC} + \mathsf{R}$ and $\mathsf{AC} + \mathsf{Sim}$ behave the same. We call $\mathsf{S} + \mathsf{LAC} + \mathsf{R}$ a secure *and live* implementation of AC if there exists a *liveness preserving* simulator Sim such that $\mathsf{S} + \mathsf{LAC} + \mathsf{R}$ and $\mathsf{AC} + \mathscr{Z}$ behave the same.

4

Consistent Communication

Contents

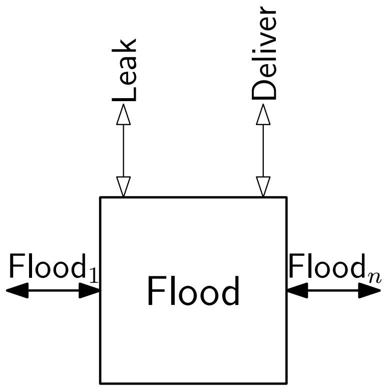
Attack Model *In this chapter we assume that*

1. *All processes are correct.*
2. *The adversary cannot drop or inject messages.*
3. *The adversary is allowed to delay and reorder messages.*
4. *The parties do not have access to clocks.*

4.1 Flooding Network

In many types of networks, for instance gossip networks, messages can overtake each other. This means that you might receive a message A which depends on another message B before you receive B. It might also be you don't receive messages from P_1 in the order in which they were sent by P_1. These behaviours might lead to inconsistencies in how the system behaves. In this section we take a brief look at techniques for ensuring consistent communication.

We assume that we have access to a so-called flooding system which guarantees that if a correct party sends a message then it is eventually delivered at all correct processes. The reason for the name ”Flooding Network” is that a natural way to ensure this property is to make sure the system is “flooded” with every new message, for instance by having each process pass on the message to anyone who has not heard it before, see also Section 2.9. We also assume that the channel is



Ports The channel connects n parties named $\mathsf{P}_1, \ldots, \mathsf{P}_n$. For each party P_i it has a port called Flood_i. It has special ports Leak and $\mathsf{Deliver}$ used to model that the network does not hide what is sent and that the adversary (or whomever controls $\mathsf{Deliver}$) can determine when messages are delivered.

Init For each P_i it keeps a queue $\mathsf{InTransit}_i$ which is initially empty.

Send On input (P_i, m) on Flood_i, it outputs (P_i, m) on Leak, and adds (P_i, m) to $\mathsf{InTransit}_j$ for $j = 1, \ldots, n$. Note that it also adds to $\mathsf{InTransit}_i$.

Deliver On input $(\mathsf{P}_i, \mathsf{P}_j, m)$ on $\mathsf{Deliver}$ where $(\mathsf{P}_i, m) \in \mathsf{InTransit}_j$, it removes (P_i, m) from $\mathsf{InTransit}_j$ and outputs (P_i, m) on Flood_j.

Figure 4.1 Flooding Network

authenticated such that the receiver knows who sent a message. We model that by outputting the name of the sender along with the message and by not allowing the adversary to inject messages. We assume no other guarantees, so messages might be arbitrarily delayed and may be reordered. We look at how to design protocols on top of this with progressively stronger consistency guarantees.

We consider a system with the following syntax, indicating how a distributed system interacts with the rest of the world (often known as the middleware, as mentioned in the Introduction):

Send A process P_i can give input of the form (P_i, m) on the user port Flood_i. We say that P_i sent m (using Flood).

Deliver A party P_i can get output of the form (P_j, m) on the user port Flood_j. We say that the message m sent by P_j was delivered to P_i (by Flood).

We assume that the system has the following properties.

User Contract We require from the process P_i that it does not send the same m twice. When the user keeps its contract, we require that the flooding system has the following safety and liveness properties. We also require that in every finite time interval a party sends at most a finite number of messages.

Safety If a correct P_j outputs (P_i, m), then earlier P_i sent (P_i, m).

Liveness If a correct P_i sends (P_i, m), then eventually all correct P_j deliver (P_i, m).

As usual we are a bit lax with defining liveness. But most definitions will be sufficient for this section. All we have to rule out is that one party can infinitely many message in finite time. If we make no restriction it could be that party P_2 sends a message m at time 0 and it is delivered at time 1. But party P_1 sends a message at time $1 - 2^{-i}$ for $i = 1, 2, \ldots$. Then m was "eventually" delivered in physical time, but yet an infinite number of other messages arrive before m. We rule that out by assuming that sending something on Flood_i takes some fixed non-zero time. Then P_1 can only send a finite number of messages between, e.g., time 0 and 1.

For protocols described in this chapter we will always assume that all processes are correct. Later in the book we look at solutions that work even if some processes do not perform correctly, or are even malicious.

As a model of this we are going to use a variant of the authenticated channel AC for n parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$. We call the network Flood, see Fig. 4.1.

Remark 4.1 (immediate delivery versus delayed delivery) Note that when some P_i sends (P_i, m) on Flood_i, then (P_i, m) is added to $\mathsf{InTransit}_i$. This means that the Deliver command can later be used to deliver (P_i, m) to P_i on Flood_i. So when we say that a party sends a message to all parties, we also mean that it sends to itself. In particular, a process P_i sending a message m via $\mathsf{Flood}.\mathsf{Flood}_i$ needs to wait to see m come back from $\mathsf{Flood}.\mathsf{Flood}_i$ to consider it delivered locally.

Notice that according to the above, there might be a delay in time between P_i sending (P_i, m_i) and the party P_i delivering (P_i, m_i) at itself. When a message is delivered after it was sent at the sender we say that it has delayed local delivery. This might look weird, but when we discuss totally ordered broadcast later it will make sense: Sometimes the delivery of m_i has to be delayed to be able to deliver some other concurrent messages first.

For the simpler network resources like Flood and FIFO, one could imagine an implementation where a party delivers locally at the same time as one sends the message to the others. Then messages sent via Flood would have immediate local delivery. It might, however, be that for some reason an implementation delivers messages locally with some delay, maybe a scheduler switches control to another thread than the one trying to immediately output the message locally. We will therefore for safety assume that all messages might experience delayed local delivery.

In Fig. 4.2 there is an example where P_1 sends m_1 via Flood and only receives m_1 after some delay. When we do later figures we will assume that Flood and the later FIFO network resource both have immediate delivery. To not unnecessarily clutter future diagrams we will assume that Flood and FIFO has immediate local delivery.

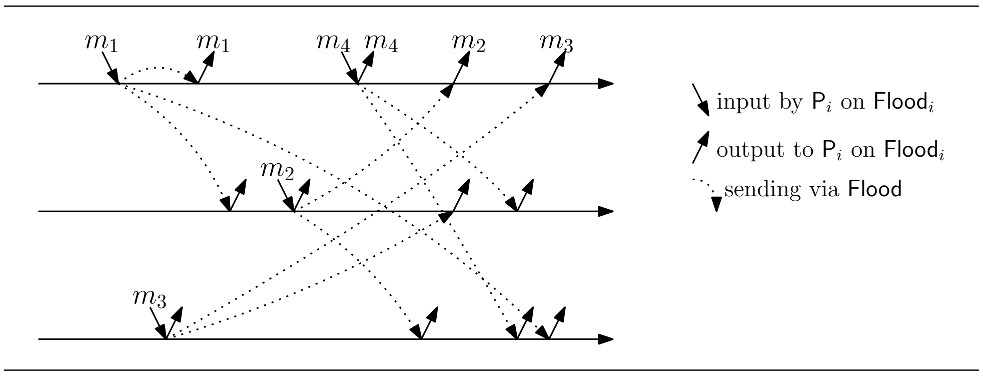

m_1
m_1
m_4
m_4
m_2
m_3
m_2
m_3
input by P_i on Flood_i
output to P_i on Flood_i
sending via Flood

Figure 4.2 Messages from P_1 arrive in different order at P_3 than the order in which they were sent. At P_1 the message m_1 has delayed local delivery, whereas m_4 has immediate local delivery.

4.2 First In, First Out

We first look at how to guarantee FIFO, i.e., we want to ensure that the communication preserves the order in which messages are sent by individual parties. We can formulate this as a safety property as follows:

FIFO If a correct P_i sends (P_i, m) and later sends (P_i, m'), then it holds for all correct P_j that if they deliver (P_i, m'), then they delivered (P_i, m) earlier.

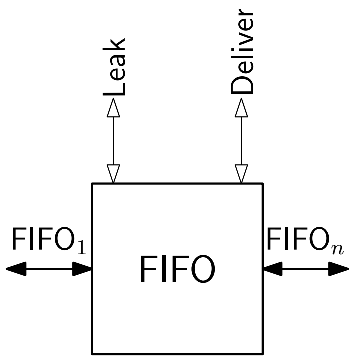
The User Contract and Liveness is as for Flood.

We model a FIFO network using a network resource/ ideal functionality. See Fig. 4.3. IT is simply designed to behave as Flood except that it keeps the FIFO safety property true.

Consider the following protocol.

1. Initially each party P_i will initialize a counter $c_i = 0$. This counter keeps track of how many messages the party P_i has already sent. Moreover each party P_i also initializes n counters $r_{i,j} = 0$ for $j = 0, \ldots, n$. These counters keep track of how many messages party P_i has received from party P_j.
2. P_i: When sending a message m, let $c_i = c_i+1$ and send (P_i, c_i, m) on the flooding network. This mean that we tag each message with it sequence number, starting from 1.
3. P_i: When receiving (P_j, c, m) store it in a buffer until $r_{i,j} = c + 1$. Then let $r_{i,j} = r_{i,j} + 1$ and output (P_j, m). This makes sense as $r_{i,j}$ is the number of messages P_i received from P_j so if $r_{i,j} = c + 1$, then (P_j, c, m) is the next message.

Clearly, if all parties are correct, this protocol ensures that all messages from each P_i are delivered by all other parties in the order in which they were sent by P_i. In particular, Flood2FIFO keeps the contract that each process sends each message at most once. This is ensured by the counter which is added to the messages sent.



Ports The channel connects n parties named $\mathsf{P}_1, \ldots, \mathsf{P}_n$. For each party P_i it has a port called FIFO_i. It has a special ports Leak and Deliver.
Init For each P_i it keeps a queue $\mathsf{InTransit}_i$ which is initially empty.
Send On input (P_i, m) on Flood_i, it outputs (P_i, m) on Leak, adds (P_i, m) to the queue $\mathsf{InTransit}_j$.
Deliver On input $(\mathsf{P}_i, \mathsf{P}_j, m)$ on Deliver, where $(\mathsf{P}_i, m) \in \mathsf{InTransit}_j$ and (P_i, m) is the element of the form $(\mathsf{P}_i, \cdot) \in \mathsf{InTransit}_j$ which is closest to the head of the queue, remove (P_i, m) from $\mathsf{InTransit}_j$ and output (P_i, m) on Flood_j.

Figure 4.3 FIFO Network

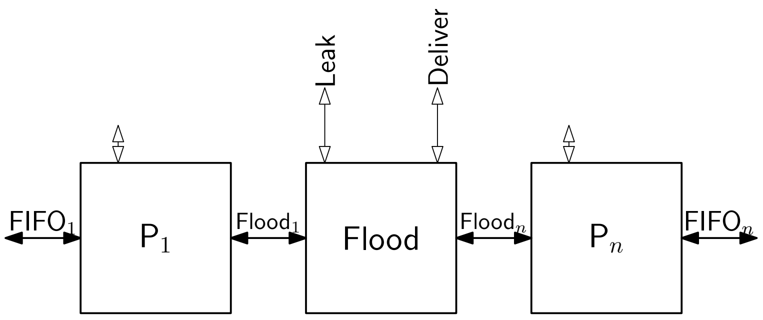
Therefore Flood will behave according to specification and eventually deliver the messages to all receivers exactly once. And it will not deliver any other messages. The logic of the counters then makes sure to deliver the messages in the right order. This ensure that Flood + Flood2FIFO securely implements FIFO

*4.2.1 A Formal Simulation Argument**

Above we gave the intuition why Flood + Flood2FIFO securely implements FIFO. Namely, whatever behaviour Flood + Flood2FIFO is capable is included in the possible behaviour of FIFO. To argue this formally one would have to argue that for any possible behaviour of Flood + Flood2FIFO there is a way to use the special ports of FIFO to make it have the same behaviour. We show the proof below with all details explained, as an example of what such a proof looks like. In the rest of the book, we will not be this formal, but the interested reader can construct such proofs along the lines of the example.

Theorem 4.2 *Let* $\mathsf{Flood2FIFO} = \{\mathsf{P}_1, \ldots, \mathsf{P}_n\}$ *where* P_i *is defined in Fig. 4.4. Then* $\mathsf{Flood} + \mathsf{Flood2FIFO} \sqsubseteq \mathsf{FIFO}$*, i.e., the protocol* Flood2FIFO *when run with the network resource* Flood *behaves as the network resource* FIFO*.*

Proof Recall that $\mathsf{Flood} + \mathsf{Flood2FIFO} \sqsubseteq \mathsf{FIFO}$ is defined to mean that there exists a simulator Sim such that Flood + Flood2FIFO and Sim + FIFO behave the same. This in turn means that for any IA, called the environment $\mathcal{Z}$, that could



Each P_i runs the following activation rules:

Init Process P_i creates a set $\mathsf{InTransit}_j = \emptyset$ for each P_j and a counter $c_i = 0$. It also for all P_j initializes a counter $r_{i,j} = 0$. The set $\mathsf{InTransit}_j$ holds the messages received on the network API from P_j. Note that P_i is also creating a set $\mathsf{InTransit}_i$ and a counter $r_{i,i}$ for itself.

Send On (P_i, m) on FIFO_i, let $c_i = c_i + 1$, and send (P_i, c_i, m) on Flood_i.

Receive On (P_j, c, m) on Flood_i add (P_j, c, m) to $\mathsf{InTransit}_j$.

Deliver If there is some P_j and some message m such that $(\mathsf{P}_j, r_{i,j} + 1, m) \in \mathsf{InTransit}_j$, then remove $(\mathsf{P}_j, r_{i,j} + 1, m)$ from $\mathsf{InTransit}_j$, let $r_{i,j} = r_{i,j} + 1$, and output (P_j, m) on FIFO_i.

Figure 4.4 The Flood2FIFO protocol

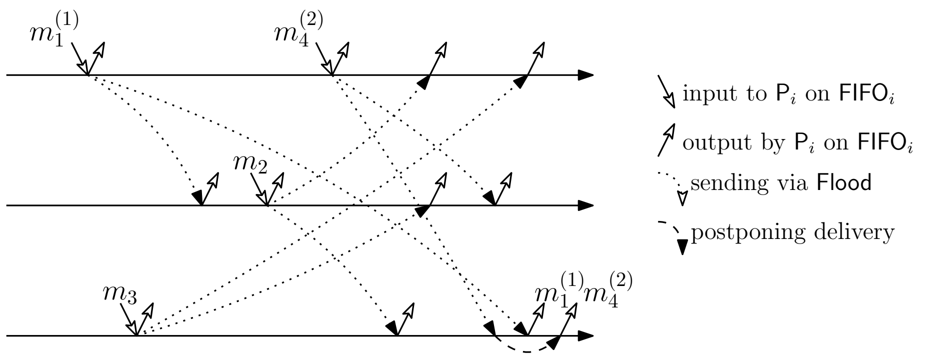


Figure 4.5 Messages from P_1 arrive in different order at P_3 than the order in which they were sent, but a counter is used to withhold m_4 until after m_1 arrives.

connect to the systems, they will give the same outputs on the same input. The setup is illustrated here:

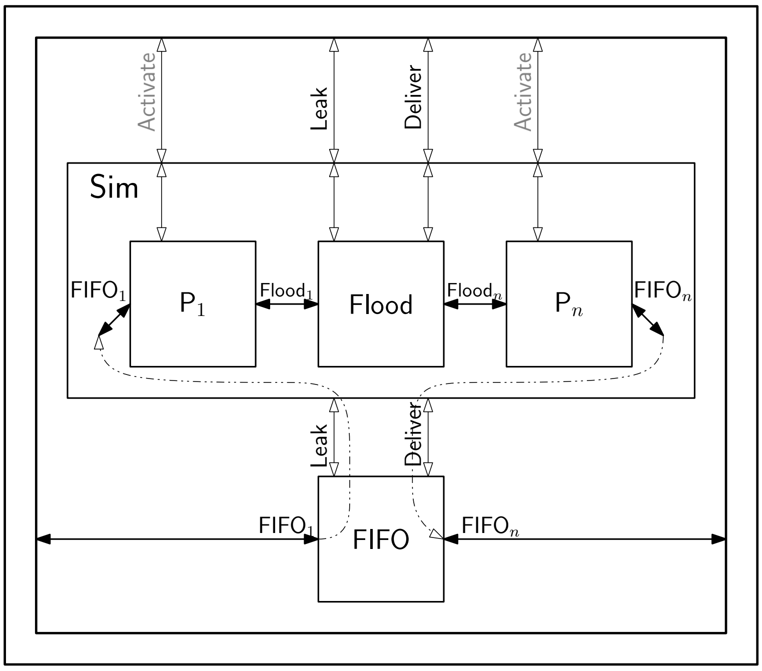
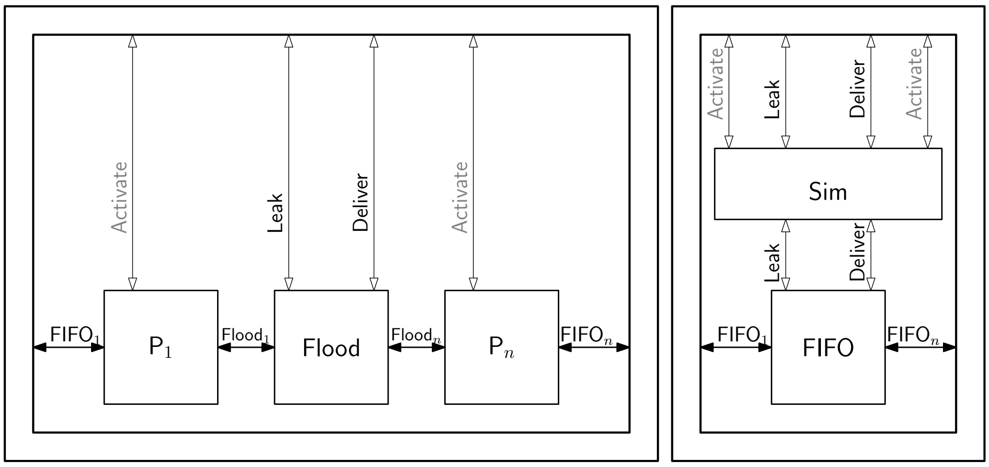


Figure 4.6 Illustration of simulating just by running the protocol. This is possible whenever the ideal functionality being implemented leaks all inputs.



We call the configuration $\mathcal{Z}$ + Flood2FIFO + Flood on the left the real world and we call the configuration $\mathcal{Z}$ + Sim + FIFO on the right the simulation.

We can make a very simple simulator Sim which basically just internally runs a copy of Flood + Flood2FIFO, called the simulated protocol. This setup is illustrated in Fig. 4.6.

When in $\mathcal{Z} + \mathsf{Sim} + \mathsf{FIFO}$ the environment inputs a message on FIFO_i, it is by design given to the simulator Sim on leak, so the simulator can just input it on $(\mathsf{Flood} + \mathsf{Flood2FIFO}).\mathsf{FIFO}_i$ in the simulated protocol. In addition the simulator will route messages to and from Flood.leak and $\mathcal{Z}$.leak. Similarly for Deliver and the activation ports of the processes in the simulated protocol. Therefore the simulated protocol $\mathsf{Flood} + \mathsf{Flood2FIFO}$ is run on exactly the same sequence of inputs as the real world protocol would have been when interacting with $\mathcal{Z}$. Finally, when the simulated protocol outputs (P_i, m) on $(\mathsf{Flood} + \mathsf{Flood2FIFO}).\mathsf{FIFO}_j$ the simulator will input $(\mathsf{P}_i, \mathsf{P}_j, m)$ on FIFO.Deliver to try to make FIFO also output (P_i, m) on FIFO_j. Notice that if this always makes FIFO output (P_i, m) on FIFO_j, then the simulation would be perfect: $\mathcal{Z}$ would always see the same input-output behaviour in the read world and the simulation.

So we only need to argue that when the simulator Sim inputs $(\mathsf{P}_i, \mathsf{P}_j, m)$ on FIFO.Deliver then FIFO will output (P_i, m) on FIFO_j. For this to be the case, we need that 1) $(\mathsf{P}_i, m) \in \mathsf{FIFO}.\mathsf{InTransit}_j$ and 2) that (P_i, m) is the message of the form $(\mathsf{P}_i, \cdot) \in \mathsf{FIFO}.\mathsf{InTransit}_j$ closest to the head of $\mathsf{FIFO}.\mathsf{InTransit}_j$. When this is that case the network resource FIFO will indeed deliver (P_i, m) on FIFO_j, as desired. To see that the two conditions hold, note that when Sim inputs $(\mathsf{P}_i, \mathsf{P}_j, m)$ on FIFO.Deliver it is because to just saw P_j output (P_i, m) in the simulated protocol. This means that just before P_j did so, it was the case that $(\mathsf{P}_i, \cdot, m) \in \mathsf{P}_j.\mathsf{InTransit}_j$. It is easy to see that this implies that $(\mathsf{P}_i, m) \in \mathsf{FIFO}.\mathsf{InTransit}_j$, as we make the simulated protocol and FIFO send and deliver the same messages at the same times. It is in addition easy to see that $(\mathsf{P}_i, m) \in \mathsf{FIFO}.\mathsf{InTransit}_j$ is the message of the form $(\mathsf{P}_i, \cdot) \in \mathsf{FIFO}.\mathsf{InTransit}_j$ closest to the head of $\mathsf{FIFO}.\mathsf{InTransit}_j$. This is because the messages in FIFO are added to $\mathsf{InTransit}_j$ in the order they were sent by P_i and because Flood2FIFO delivers them on FIFO_j in the order they were sent by P_i. □

Clearly the above proof strategy is very generic. If we implement a network resource which leaks all inputs on leak, then we can simulate simply by running the real world protocol. All that is needed is then that the protocol implements the safety properties of of the network resource that we implement. When this is the case, the simulator can simply instruct the network resource to give the same outputs as the simulated protocol. In the following we will therefore only argue that our protocols implement the desired safety properties. We will not in each proof go through the above tedious simulation argument.

4.3 Causality

When using FIFO as network resource it might still happen that the messages from different parties do not arrive in a meaningful order. Consider the messages m_1 and m_2 in Fig. 4.5. When P_2 sent m_2 it already delivered m_1. That means that the application layer using FIFO via $\mathsf{FIFO}.\mathsf{FIFO}_2$ might have chosen m_2 as a function of m_1. We say that m_1 could have caused m_2. For instance m_1 could be a question in some chat forum and m_2 could be the answer. But notice that at

P_3 first m_2 is delivered, and then m_1. It might confuse the application users at P_3 to read the answer before the question. In this section we will focus on defining and implementing a network resource, Causal, which ensures that messages are delivered in a causal order: if a message m_1 could have caused m_2, then all parties deliver m_1 before m_2.

4.3.1 An Implementation

We will now fall into the trap of giving the answer before the question. We will first precent a protocol FIFO2Causal which given a FIFO network resource implements a Causal network resource. After that we will then define what is the Causal network resource. The reason why we do it this way around is that the protocol is actually pretty simple and almost self explanatory. The definition is causality on the other hand is a bit subtle. So let us start with the concrete and simple and then lift ourselves up to the abstract and complicated afterward.

Before describing the protocol it is convenient to have a notion of vector clock. For a system with n parties $\mathsf{P}_1, \dots, \mathsf{P}_n$ a vector clock is just a vector with n natural number (one for each P_i), i.e., $\mathsf{VectorClock} \in \mathbb{N}^n$. We use $\mathsf{VectorClock}[\mathsf{P}_j] \in \mathbb{N}$ to denote the entry associated by P_j. Each party will have its own vector clock. We denote the vector clock of P_i by $\mathsf{VectorClock}(\mathsf{P}_i)$. To avoid confusion note that $\mathsf{VectorClock}(\mathsf{P}_i) \in \mathbb{N}^n$. So each party holds its own complete vector clock.

The basic idea of the vector clocks are that if $\mathsf{VectorClock}(\mathsf{P}_i)[\mathsf{P}_j] = s$, then P_i knows that P_j sent at least s messages. Furthermore, there might be more messages that P_j sent via FIFO, but these cannot have affected P_i since P_i did not receive them, neither directly or indirectly. When $\mathsf{VectorClock}(\mathsf{P}_i)[\mathsf{P}_j] = s$, then only the s first messages sent by P_j could have affected P_i.

Below follows a few simple observations that are useful in understanding the algorithm.

When P_i sends a message m it increments $\mathsf{VectorClock}(\mathsf{P}_i)[\mathsf{P}_i]$ by one, as P_i just learned that P_i sent a message.

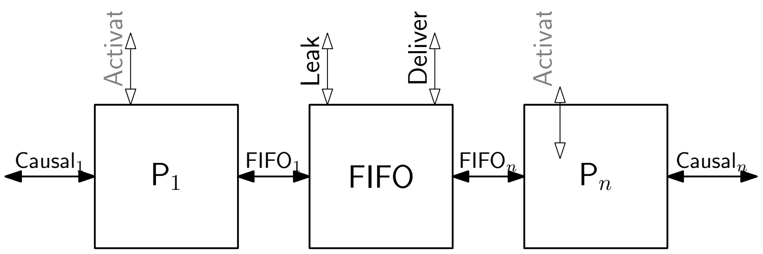
When P_i sends a message m its sends along $\mathsf{VectorClock}(\mathsf{P}_i)$. When P_i sends the message m that message can be influenced by exactly the messages that could have influenced P_i at the time P_i sends the message. So sending along the vector clock annotates the message m with a vector clock $\mathsf{VectorClock}(\mathsf{P}_i, m) = \mathsf{VectorClock}(\mathsf{P}_i)$ which allows to determine which messages could have influenced m. Namely, only the first $s = \mathsf{VectorClock}(\mathsf{P}_i, m)[\mathsf{P}_j]$ messages from P_j could have influenced m. Therefore any receiver P_r knows that it is safe to deliver (P_i, m) once it delivered s messages from P_j: Using a FIFO network resources ensures that once P_r delivered s messages from P_j it is the s that could have influenced m.

To keep track of how many messages were delivered from each party, P_j keeps a vector clock $\mathsf{Delivered}(\mathsf{P}_j)$, where $\mathsf{Delivered}(\mathsf{P}_j)[\mathsf{P}_i]$ is how many messages P_j delivered from P_i. This gives the rule that P_j can deliver (P_i, m) once it holds for all P_k that

$$\mathsf{Delivered}(\mathsf{P}_j)[\mathsf{P}_k] \geq \mathsf{VectorClock}(\mathsf{P}_i, m)[\mathsf{P}_k] \ .$$

There is one exception: when receiving a message m which is message number s sent by P_i, then $\mathsf{VectorClock}(\mathsf{P}_i, m)[\mathsf{P}_i] = s$, as the vector clock of m counts the sending of m itself. So, for instance, if m is the first message sent by P_i, then $\mathsf{VectorClock}(\mathsf{P}_i, m)[\mathsf{P}_i] = 1$. But since P_j did not receive m yet it might be the case that $\mathsf{Delivered}(\mathsf{P}_j)[\mathsf{P}_i] = s - 1$. And when $\mathsf{Delivered}(\mathsf{P}_j)[\mathsf{P}_i] = s - 1$ we would actually like to deliver m at P_j, as it is the next message from P_i. We therefore allow to deliver a message from P_i already when

$$\mathsf{Delivered}(\mathsf{P}_j)[\mathsf{P}_i] \geq \mathsf{VectorClock}(\mathsf{P}_i, m)[\mathsf{P}_i] - 1 \ .$$



All parties P_i run the following activation rules

Init P_i: Initially let $\mathsf{VectorClock}(\mathsf{P}_i) = \mathbf{0}$ and let $\mathsf{Delivered}(\mathsf{P}_i) = \mathbf{0}$.

Send On input (P_i, m) on Causal_i, let $\mathsf{VectorClock}(\mathsf{P}_i) = \mathsf{VectorClock}(\mathsf{P}_i) + \mathsf{P}_i$ and $\mathsf{VectorClock}(\mathsf{P}_i, m) = \mathsf{VectorClock}(\mathsf{P}_i)$. Then send $(\mathsf{P}_i, m, \mathsf{VectorClock}(\mathsf{P}_i, m))$ on FIFO_i.

Deliver On $(\mathsf{P}_j, m, \mathsf{VectorClock}(\mathsf{P}_j, m))$ on FIFO_i at P_i, where $\mathsf{VectorClock}(\mathsf{P}_j, m) \leq \mathsf{Delivered}(\mathsf{P}_i) + \mathsf{P}_j$, deliver (P_j, m) on Causal_i and let $\mathsf{VectorClock}(\mathsf{P}_i) = \max(\mathsf{VectorClock}(\mathsf{P}_i), \mathsf{VectorClock}(\mathsf{P}_j, m))$ and $\mathsf{Delivered}(\mathsf{P}_i) = \mathsf{Delivered}(\mathsf{P}_i) + \mathsf{P}_j$.

Figure 4.7 The Protocol FIFO2Causal.

Finally, note that if P_k receives a message $(\mathsf{P}_k m)$ where for some P_j it holds that $\mathsf{VectorClock}(\mathsf{P}_k, m)[\mathsf{P}_j] > \mathsf{VectorClock}(\mathsf{P}_i)[\mathsf{P}_j]$, then m could have been influenced by more message from P_j than P_i might have been influenced by. But now these messages could have influenced P_i indirectly via m, so P_i updates $\mathsf{VectorClock}(\mathsf{P}_i)[\mathsf{P}_j] = \mathsf{VectorClock}(\mathsf{P}_k, m)[\mathsf{P}_j]$. Put more concisely, we could just always let $\mathsf{VectorClock}(\mathsf{P}_i)[\mathsf{P}_j] = \max(\mathsf{VectorClock}(\mathsf{P}_i)[\mathsf{P}_j], \mathsf{VectorClock}(\mathsf{P}_k, m)[\mathsf{P}_j])$ for all P_j when receiving a new message (P_k, m). All parties update their clocks like this in response to all incoming messages.

For some examples of how vector clocks are computed, see Fig. 4.12 appearing later in the chapter.

We now put the above observations together into an algorithm. For brevity we will use

$$\mathsf{VectorClock}' = \mathsf{VectorClock} + \mathsf{P}_i$$

to denote the vector clock $\mathsf{VectorClock}'$ for which $\mathsf{VectorClock}'[\mathsf{P}_j] = \mathsf{VectorClock}[\mathsf{P}_j]$

for all $\mathsf{P}_j \neq \mathsf{P}_i$ and $\mathsf{VectorClock}'[\mathsf{P}_i] = \mathsf{VectorClock}[\mathsf{P}_i] + 1$ for all $\mathsf{P}_j \neq \mathsf{P}_i$ and We define

$$\mathsf{VectorClock}_1 \leq \mathsf{VectorClock}_2$$

to mean that

$$\forall \mathsf{P}_k \,(\, \mathsf{VectorClock}_1[\mathsf{P}_k] \leq \mathsf{VectorClock}_2[\mathsf{P}_k]\,) \ .$$

We define

$$\mathsf{VectorClock} = \max(\,\mathsf{VectorClock}_1, \mathsf{VectorClock}_2\,)$$

to mean that

$$\forall \mathsf{P}_k \,(\, \mathsf{VectorClock}[\mathsf{P}_k] = \max(\mathsf{VectorClock}_1[\mathsf{P}_k], \mathsf{VectorClock}_2[\mathsf{P}_k])\,) \ .$$

We let $\mathbf{0}$ denote the all-0 vector, i.e., $\mathbf{0}[\mathsf{P}_j] = 0$ for all P_j. The algorithm is given in Fig. 4.7. There is an example of the execution of FIFO2Causal in Fig. 4.8

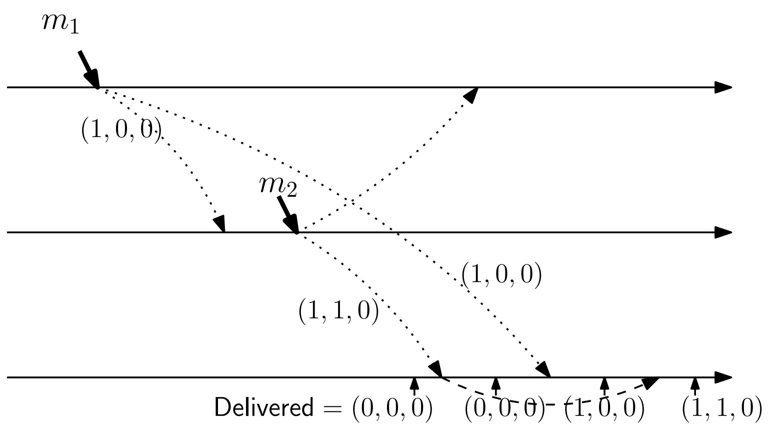


Figure 4.8 Message m_2 from P_2, which say one message m_1 from P_1, overtakes m_1. However, the counters of how many message you saw from each other party are sent along with the messages, as annotated on the dotted lines. When m_2 arrives at P_3 it has clock $(1,1,0)$. The delivery clock of P_3 is $(0,0,0)$, so it cannot deliver a message with clock $(1,1,0)$ from P_2 as the first entry of $(0,0,0)$ show that P_3 did not deliver any messages from P_1 and the first entry of $(1,1,0)$ show that P_2 *did* deliver a messages from P_1 when it sent m_2. Therefore P_3 will wait with delivering m_2. When m_1 arrives at P_3 it has clock $(1,0,0)$ and since the delivery clock of P_3 is $(0,0,0)$ it can be seen that P_3 delivered all the same messages as P_1 when it sent m_1, except m_1 itself. Therefore P_3 can output m_1 and increment its delivery clock to $(1,0,0)$. Now that the delivery clock is $(1,0,0)$ it holds for the clock $(1,1,0)$ of m_2 that it is the same as the delivery clock except it s one larger for the sender of m_2. Hence m_2 can be delivered and the delivery clock upgraded to $(1,1,0)$.

4.3.2 The Causal-Past Relation

We would now like to prove that $\mathsf{FIFO2Causal}$ implements causal communication. For this we need to make a definition of that causal communication is. We will derive this definition via the causal-past relation, which describes when events in a distributed system might be causally related. The relation is more idiomatically called the happens-before relation, but we call it the causal-past relation here as this better captures our particular use.

The causal-past relation is a binary relation $\hookrightarrow$ on messages (P, m). We will write it as $(\mathsf{P}_i, m_i) \hookrightarrow (\mathsf{P}_j, m_j)$. The intuitive meaning of this is that, in a particular run of the system, m_j may depend on m_i, for instance because P_j received m_i before it sent m_j. Note that the fact that P_j received m_i before sending m_j does not necessarily mean that m_j in fact depends on m_i, only that it might depend on m_i. Maybe m_j is not an answer to a question posed in m_i. Perhaps m_j is a message on a completely unrelated topic. We cannot say anything about the actual casual relationship between m_j and m_i without knowing more details about the application we are designing. Therefore, as usual, to be on the safe side, we say that if there is a chance that m_j depends on m_i, then we should treat m_j as if it in fact was causally related to m_i. Note also that not all send events are necessarily related by the causal-past relation, and that two send events can also be independent from each other, in which case they are not in the relation, and it does not make sense to ask if one happens before the other (since communication takes time, P_j might have sent m_j before it has seen m_i, but after m_i was sent by P_i).

We use $\mathsf{CausalPast}(\mathsf{P}_j, m_j)$ to denote the set of of (P_i, m_i) on which (P_j, m_j) may depend, i.e.,

$$\mathsf{CausalPast}(\mathsf{P}_j, m_j) = \{ (\mathsf{P}_i, m_i) \mid (\mathsf{P}_i, m_i) \hookrightarrow (\mathsf{P}_j, m_j) \} .$$

Similarly we define

$$\mathsf{CausalFuture}(\mathsf{P}_j, m_j) = \{ (\mathsf{P}_i, m_i) \mid (\mathsf{P}_j, m_j) \hookrightarrow (\mathsf{P}_i, m_i) \} .$$

Clearly, if we know the set $\mathsf{CausalPast}(\mathsf{P}_j, m_j)$, then we are also able evaluate the $\hookrightarrow$ relation for the given message, since it holds that

$$(\mathsf{P}_i, m_i) \hookrightarrow (\mathsf{P}_j, m_j) \iff (\mathsf{P}_i, m_i) \in \mathsf{CausalPast}(\mathsf{P}_j, m_j) . \tag{4.1}$$

So, if we can define $\mathsf{CausalPast}(\mathsf{P}_j, m_j)$ for all (P_j, m_j), then we also defined $(\mathsf{P}_i, m_i) \hookrightarrow (\mathsf{P}_j, m_j)$ for all (P_i, m_i) and (P_j, m_j), and *vice versa.*

For an execution of a given protocol between a set of parties, we can determine the relation $\hookrightarrow$ for all messages by determining all the sets $\mathsf{CausalPast}(\mathsf{P}_j, m_j)$. We can do so operationally, going through the run of the protocol step by step, as described in the method below. The method described keeps track of auxiliary sets $\mathsf{CausalPast}(\mathsf{P}_i)$ for each party. These sets contain, at any given time, all messages that may have influenced P_i's behaviour up to now. Below we use P_i to denote the process at the application layer using the causal network. In send on Causal_i and is delivered messages on Causal_i. We keep track of the message sent in the causal network and their causal pasts. It is fairly easy so see that the following is a good operational definition.

1. Initially let $\mathsf{CausalPast}(\mathsf{P}_i) = \emptyset$ for all P_i.

   Justification: Initially no messages can have affected P_i.
2. On input (P_i, m) on Causal_i at P_i, let $\mathsf{CausalPast}(\mathsf{P}_i) = \mathsf{CausalPast}(\mathsf{P}_i) \cup \{(\mathsf{P}_i, m)\}$ and let $\mathsf{CausalPast}(\mathsf{P}_i, m) = \mathsf{CausalPast}(\mathsf{P}_i)$.

   Justification: When (P_i, m) is input on Causal_i at P_i, then clear m now could have affected P_i, as it was P_i that input m to the system, so we let $\mathsf{CausalPast}(\mathsf{P}_i) = \mathsf{CausalPast}(\mathsf{P}_i) \cup \{(\mathsf{P}_i, m)\}$. Furthermore, since m was input at P_i all the messages that could have affected P_i could have affected m, so we let $\mathsf{CausalPast}(\mathsf{P}_i, m) = \mathsf{CausalPast}(\mathsf{P}_i)$.
3. When (P_j, m) is delivered on Causal_i at P_i, let $\mathsf{CausalPast}(\mathsf{P}_i) = \mathsf{CausalPast}(\mathsf{P}_i) \cup \mathsf{CausalPast}(\mathsf{P}_j, m)$.

   Justification: When m is delivered on Causal_i at P_i then the application layer connected to Causal_i might now be affected by m and therefore indirectly also the messages that might have influence m. Therefore we set $\mathsf{CausalPast}(\mathsf{P}_i) = \mathsf{CausalPast}(\mathsf{P}_i) \cup \mathsf{CausalPast}(\mathsf{P}_j, m)$.

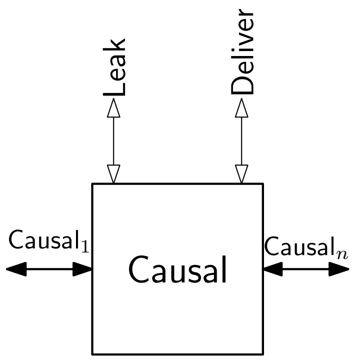
As mentioned, the causal-past relation $(\mathsf{P}_i, m_1) \hookrightarrow (\mathsf{P}_j, m_2)$ is supposed to capture when m_1 might have influenced the contents of m_2. The first rule says that initially no party could have been influenced by any message. This is clear as no messages were sent so far.

The second rule says that if P_i sends a message, then the set of messages that might have influenced P_i grows by this message. This is clear as P_i has certainly seen the message. Furthermore, the message might have been influenced by all messages that may have influenced P_i – simply because the message was produced by the system running at P_i's site.

The third rule says that if P_i has delivered a message m, then P_i may now have been influenced by all messages that may have influenced m.

The last implicit rule postulates that there is no other way m_2 could depend on m_1. This rule is implicit in the sense that if we thought there were other ways in which messages could affect each other we would have added more explicit update rules. This fourth rule is true in a fully asynchronous system, where parties are not allowed to use a clock and where in general information only flows via message passing inside our flooding network. But the rule is not always true. [1]

[1] Assume that m_1 is a bit. Consider the case where P_i sends m_1 if $m_1 = 1$ and sends nothing when $m_1 = 0$. Assume that P_j knows when m_1 is sent and knows an upper bound on the delivery time. Then when after long enough it did not receive any message from P_i party P_j it can set $m_2 = 0$. If it does receive some m_1 it sets $m_2 = 1$. So clearly m_2 always depends on m_1, but in half of the cases no message was sent from P_i to P_j so by definition we have that $(\mathsf{P}_i, m_1) \not\hookrightarrow (\mathsf{P}_j, m_2)$ in those cases. The problem is that if parties are allowed to use time, then information can flow in other ways than via message passing. We will later look at other unfortunate ways information can flow if for instance parties are on the same machine. It could also be that even though no message is sent from P_i to P_j between P_i sending m_i and P_j sending m_j, they could have communicated by other means, maybe by sitting in the same



Ports The channel connects n parties named $\mathsf{P}_1, \dots, \mathsf{P}_n$. For each party P_i it has a port called Causal_i. It has a special ports Leak and $\mathsf{Deliver}$.

Init For each P_i it keeps a set $\mathsf{Delivered}_i$ which is initially empty. Initialize a set Sent which is initially empty.

Send On input (P_i, m) on Causal_i, output (P_i, m) on leak, let $\mathsf{Sent} = \mathsf{Sent} \cup \{(\mathsf{P}_i, m)\}$, let $\mathsf{CausalPast}(\mathsf{P}_i) = \mathsf{Causal}(\mathsf{P}_i) \cup \{(\mathsf{P}_i, m)\}$, and let $\mathsf{CausalPast}(\mathsf{P}_i, m) = \mathsf{CausalPast}(\mathsf{P}_i)$.

Deliver On $(\mathsf{P}_j, \mathsf{P}_i, m)$ on $\mathsf{Deliver}$, where $(\mathsf{P}_j, m) \in \mathsf{Sent}$, $(\mathsf{P}_j, m) \notin \mathsf{Delivered}_i$, and $\mathsf{CausalPast}(\mathsf{P}_j, m) \subseteq \mathsf{Delivered}_i \cup \{(\mathsf{P}_j, m)\}$, deliver (P_j, m) on Causal_i and let $\mathsf{CausalPast}(\mathsf{P}_i) = \mathsf{CausalPast}(\mathsf{P}_i) \cup \mathsf{CausalPast}(\mathsf{P}_j, m)$ and $\mathsf{Delivered}(\mathsf{P}_i) = \mathsf{Delivered}(\mathsf{P}_i) \cup \{(\mathsf{P}_j, m)\}$.

Figure 4.9 Causal Network

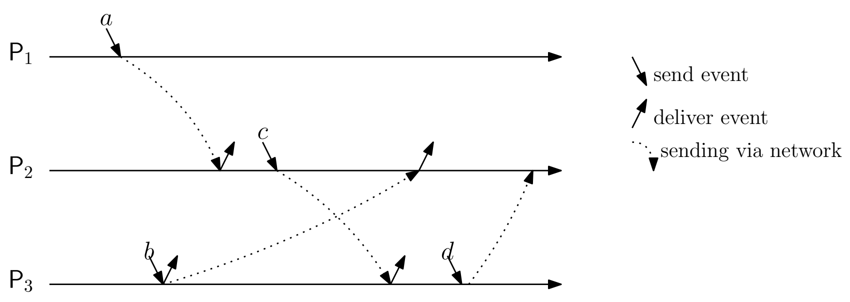


Figure 4.10 Causal Past Example

Example 4.3 Let us look at a concrete small example. Suppose the following events happen:

- P_1 sends message a (send-event (P_1, a) happens at P_1).
- P_3 sends message b (send-event (P_3, b) happens at P_3.

room and talking. We cannot take care of that in the definition, as we only have access to the communication inside the system.

- Message (P_1, a) arrives at P_2. (receive-event (P_1, a) happens at P_2)
- P_2 sends message c to P_3.
- Message (P_3, b) arrives at P_2 .
- Message (P_2, c) arrives at P_3.
- P_3 sends message d, which arrives at P_2.

See Fig. 4.10. In this example, we have 4 send-events. First, (P_1, a) and (P_3, b) are independent, the two involved parties have not talked to each other yet, so there is no way a could depend on b or vice versa. Second, we have $(\mathsf{P}_1, a) \hookrightarrow (\mathsf{P}_2, c)$ since P_2 gets a before it sends c. And finally, (P_3, b) and (P_2, c) are independent, because P_2 produces c before it receives b. △

Exercise 4.1 (Casusal Past) Recall the series of events we considered in Example 4.3

- P_1 sends message a. (send-event (P_1, a) happens).
- P_3 sends message b.
- Message (P_1, a) arrives at P_2.
- P_2 sends message c.
- Message (P_3, b) arrives at P_2.
- Message (P_2, c) arrives at P_3.
- P_3 sends message d, which arrives at P_2.

Assume players use the above protocol based on vector clocks to communicate. For each send-event, specify the vector clock that accompanies the message that is sent. There are 4 messages sent, and hence 6 pairs of messages. For each such pair, use the vector clocks associated to the messages to determine if the messages in the pair are concurrent, or the pair is in the causal past relation.

Exercise 4.2 (properties of causality sets) Prove that the causal-past set and the causal-future set have the following properties:

Set Equivalence: It holds that

$$(\mathsf{P}_i, m_i) \hookrightarrow (\mathsf{P}_j, m_j)$$

if and only if

$$\mathsf{CausalPast}(\mathsf{P}_i, m_i) \subseteq \mathsf{CausalPast}(\mathsf{P}_j, m_j) \ .$$

Note that this does not follow by definition, as the definition says that $(\mathsf{P}_i, m_i) \hookrightarrow (\mathsf{P}_j, m_j)$ if and only if $(\mathsf{P}_i, m_i) \in \mathsf{CausalPast}(\mathsf{P}_j, m_j)$. So it is enough to prove that $(\mathsf{P}_i, m_i) \in \mathsf{CausalPast}(\mathsf{P}_j, m_j)$ if and only if $\mathsf{CausalPast}(\mathsf{P}_i, m_i) \subseteq \mathsf{CausalPast}(\mathsf{P}_j, m_j)$.

Transitive: $\mathsf{CausalPast}(\mathsf{P}, m) \cap \mathsf{CausalFuture}(\mathsf{P}, m) = \{(\mathsf{P}, m)\}$.

Exercise 4.3 (properties of causal-past relation) Prove that the causal-past relation $\hookrightarrow$ has the following properties:

Transitive: If $(\mathsf{P}_i, m_i) \hookrightarrow (\mathsf{P}_j, m_j)$ and $(\mathsf{P}_j, m_j) \hookrightarrow (\mathsf{P}_k, m_k)$, then $(\mathsf{P}_i, m_i) \hookrightarrow (\mathsf{P}_k, m_k)$.

Reflexive: For all messages it holds that $(\mathsf{P}_i, m_i) \hookrightarrow (\mathsf{P}_i, m_i)$.

Antisymmetric: If $(\mathsf{P}_i, m_i) \hookrightarrow (\mathsf{P}_j, m_j)$ and $(\mathsf{P}_j, m_j) \hookrightarrow (\mathsf{P}_i, m_i)$, then $(\mathsf{P}_i, m_i) = (\mathsf{P}_j, m_j)$.

4.3.3 The Causal Communication Network Resource, Causal

With the above definition of the causal-past relation it is simple to construct an ideal functionality Causal which only allows causal communication. If will work in the same general way as FIFO, but it will only allow message to be delivered in causal order. To ensure this it keeps track of which messages were delivered at each party P_i. For this Causal uses a set $\mathsf{Delivered}_i$. Now before delivering a message m from P_j at P_i it simply checks whether $\mathsf{CausalPast}(\mathsf{P}_j, m) \subseteq \mathsf{Delivered}_i$. If so, then all messages that (P_j, m) could depend on were already delivered at P_i, so it is safe to deliver (P_j, m) at P_i. The condition $\mathsf{CausalPast}(\mathsf{P}_j, m) \subseteq \mathsf{Delivered}_i$ is, however, too strict. Recall that $(\mathsf{P}_j, m) \in \mathsf{CausalPast}(\mathsf{P}_j, m)$. And clearly, before we deliver (P_j, m) at P_i we will have that $(\mathsf{P}_j, m) \notin \mathsf{Delivered}(\mathsf{P}_i)$. So, clearly $\mathsf{CausalPast}(\mathsf{P}_j, m) \not\subseteq \mathsf{Delivered}_i$. So the delivery of (P_j, m) is deadlocked, waiting for the delivery of itself. So what we want to say is that $\mathsf{CausalPast}(\mathsf{P}_j, m) \subseteq \mathsf{Delivered}_i$, except that it is allowed that (P_j, m) is not in $\mathsf{Delivered}_i$, which we can say by requiring that $\mathsf{CausalPast}(\mathsf{P}_j, m) \subseteq \mathsf{Delivered}_i \cup \{(\mathsf{P}_j, m)\}$. With this in mind, the network resource Causal is defined as in Fig. 4.9.

Recall that we will in general not do full simulation proof, just argue that out protocols have the desired safety properties. Note that Causal behaves exactly as FIFO except that it delivers messages according to the following safety property.

Causal Order If $(\mathsf{P}_i, m) \hookrightarrow (\mathsf{P}_j, m')$ and $(\mathsf{P}_i, m) \neq (\mathsf{P}_j, m')$, then it holds for all correct P_k that if they deliver (P_j, m'), then they have previously delivered (P_i, m).

The User Contract and Liveness properties are as for Flood.

It can be seen that Causal is the least restrictive ideal functionality guaranteeing this safety property. Let us argue one of the directions, namely that Causal maintains the above safety property. To prove this we have to assume that $(\mathsf{P}_i, m) \hookrightarrow (\mathsf{P}_j, m')$ and $(\mathsf{P}_i, m) \neq (\mathsf{P}_j, m')$ and that P_k delivered (P_j, m'). Then we have to prove that P_k previously delivered (P_i, m).

To see this, note that $\mathsf{Delivered}_i$ is exactly the messages delivered at P_k. So P_k having delivered (P_j, m') is the same as $(\mathsf{P}_j, m') \in \mathsf{Delivered}_k$. At the point where Causal added (P_j, m') to $\mathsf{Delivered}_k$ it check that

$$\mathsf{CausalPast}(\mathsf{P}_j, m') \subseteq \mathsf{Delivered}_k \cup \{(\mathsf{P}_j, m')\} \ . \tag{4.2}$$

Since $(\mathsf{P}_i, m) \hookrightarrow (\mathsf{P}_j, m')$ we have by definition that $(\mathsf{P}_i, m) \in \mathsf{CausalPast}(\mathsf{P}_j, m')$. So from Eq. 4.2 we get that

$$(\mathsf{P}_i, m) \in \mathsf{Delivered}_k \cup \{(\mathsf{P}_j, m')\} \ . \tag{4.3}$$

Since $(\mathsf{P}_i, m) \neq (\mathsf{P}_j, m')$ it follows from Eq. 4.3 that

$$(\mathsf{P}_i, m) \in \mathsf{Delivered}_k \ .$$

Therefore (P_i, m) was already delivered before the point where (P_j, m') was delivered.

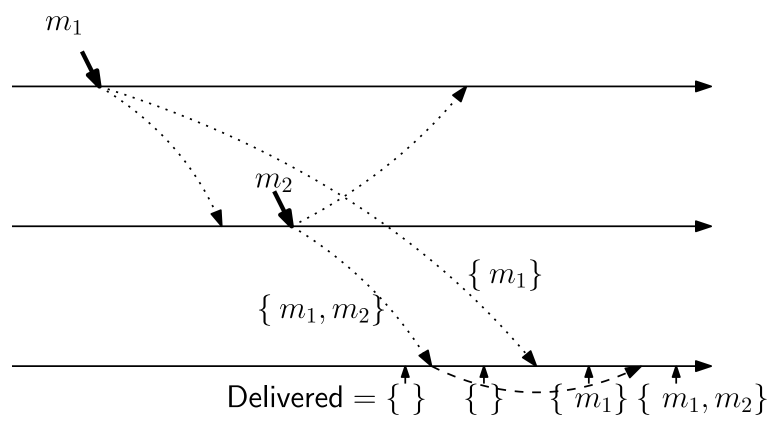


Figure 4.11 Message m_2 from P_2, which has m_1 in its causal past, overtakes m_1. However, the causal past of messages are sent along with them, as annotated on the dotted lines. When m_2 arrives at P_3 it has causal past $\{m_1, m_2\}$ and P_3 has $\mathsf{Delivered} = \emptyset$, so m_2 cannot be delivered. Later when m_1 arrives at P_3 it has causal past $\{m_1\}$ and P_3 has $\mathsf{Delivered} = \emptyset$, so m_1 can be delivered. After this m_2 still has causal past $\{m_1, m_2\}$ but P_3 now has $\mathsf{Delivered} = \{m_1\}$, so not m_2 can be delivered. After this P_3 has $\mathsf{Delivered} = \{m_1, m_2\}$.

4.3.4 Proving Security

We now prove the following theorem.

Theorem 4.4 $\mathsf{FIFO2Causal} + \mathsf{FIFO} \sqsubseteq \mathsf{Causal}$.

Since we are not going full simulation proofs, we just have to argue that $\mathsf{FIFO2Causal} + \mathsf{FIFO}$ produces outputs according to the safety property Causal Consistency defined above. As a step towards this, we first give an implementation for which this is trivial. The protocol we give is very inefficient but trivially a safe protocol. It will simply keep track of which messages were delivered and then only deliver a message m if all the messages that may have influenced m have already been delivered. All parties P_i run the following code:

Init Initially let $\mathsf{CausalPast}(\mathsf{P}_i) = \emptyset$ for all P_i and let $\mathsf{Delivered}(\mathsf{P}_i) = \emptyset$

Send On input (P_i, m) on Causal_i, let $\mathsf{CausalPast}(\mathsf{P}_i) = \mathsf{CausalPast}(\mathsf{P}_i) \cup \{(\mathsf{P}_i, m)\}$ and let $\mathsf{CausalPast}(\mathsf{P}_i, m) = \mathsf{CausalPast}(\mathsf{P}_i)$. Then send $(\mathsf{P}_i, m, \mathsf{CausalPast}(\mathsf{P}_i, m))$ on Flood_i.

Deliver On receiving $(\mathsf{P}_j, m, \mathsf{CausalPast}(\mathsf{P}_j, m))$ on Causal_i wait until $\mathsf{CausalPast}(\mathsf{P}_j, m) \subseteq$

$\mathsf{Delivered}(\mathsf{P}_i) \cup \{(\mathsf{P}_j, m)\}$. Then deliver (P_j, m) on Causal_i, and let $\mathsf{CausalPast}(\mathsf{P}_i) = \mathsf{CausalPast}(\mathsf{P}_i) \cup \mathsf{CausalPast}(\mathsf{P}_j, m)$. Also add (P_j, m) to $\mathsf{Delivered}(\mathsf{P}_i)$.

Let us call this protocol Flood2Causal. There is an example of an execution in Fig. 4.11. Note that the sets are being defined exactly as in Causal. So it is completely trivial that

Lemma 4.5 $\mathsf{Flood2Causal} + \mathsf{Flood} \sqsubseteq \mathsf{Causal}$.

4.3.5 Vector Clocks

As we have just argued, the above protocol Flood2Causal satisfies the desired properties of safety and liveness (in a pretty much straightforward way). It is, however, horribly inefficient, since every time a message m is sent, the entire set of messages that might have influenced m is sent along. We now give a much more efficient protocol. Instead of all parties keeping track of all messages received from P_j they could just use the FIFO network instead of the Flood network, and then only keep track of how many messages they received. The crucial observation is that if two parties P_i and P_j receive the same messages in the same order from P_k and they received the same number of messages, then they also received the same set of messages. So the overall idea is to replace the sets of messages from the inefficient protocol by counters.

More precisely this means the following: instead of maintaining his causal past, each P_i maintains an array $\mathsf{VectorClock}(\mathsf{P}_i)$ of integers, where $\mathsf{VectorClock}(\mathsf{P}_i)[\mathsf{P}_k]$ is the number of messages that have currently been received from P_k. When a message m_i is sent, P_i will append the current state of his vector clock to the message, this is denoted $\mathsf{VectorClock}(\mathsf{P}_i, m_i)$, and thus $\mathsf{VectorClock}(\mathsf{P}_i, m_i)[\mathsf{P}_k]$ is the number of messages P_i has received from P_k at the time he sent m_i. Moreover, instead of remembering all message that were delivered, each P_i will maintain an array $\mathsf{Delivered}(\mathsf{P}_i)$, of the same type as vector clocks, but where $\mathsf{Delivered}(\mathsf{P}_i)[\mathsf{P}_k]$ contains the number of messages from P_k that were delivered. With this in mind, we can look at Flood2Causal and FIFO2Causal next to each other:

Init

Flood2Causal: Initially let $\mathsf{CausalPast}(\mathsf{P}_i) = \emptyset$ for all P_i and let $\mathsf{Delivered}(\mathsf{P}_i) = \emptyset$

FIFO2Causal: P_i: Initially let $\mathsf{VectorClock}(\mathsf{P}_i) = \mathbf{0}$ and let $\mathsf{Delivered}(\mathsf{P}_i) = \mathbf{0}$.

Send

Flood2Causal: On input (P_i, m) on Causal_i, let $\mathsf{CausalPast}(\mathsf{P}_i) = \mathsf{CausalPast}(\mathsf{P}_i) \cup \{(\mathsf{P}_i, m)\}$ and let $\mathsf{CausalPast}(\mathsf{P}_i, m) = \mathsf{CausalPast}(\mathsf{P}_i)$. Then send $(\mathsf{P}_i, m, \mathsf{CausalPast}(\mathsf{P}_i, m))$ on Flood_i.

FIFO2Causal: On input (P_i, m) on Causal_i, let $\mathsf{VectorClock}(\mathsf{P}_i) = \mathsf{VectorClock}(\mathsf{P}_i) + \mathsf{P}_i$ and $\mathsf{VectorClock}(\mathsf{P}_i, m) = \mathsf{VectorClock}(\mathsf{P}_i)$. Then send $(\mathsf{P}_i, m, \mathsf{VectorClock}(\mathsf{P}_i, m))$ on FIFO_i.

Deliver

$\mathsf{Flood2Causal}$: On receiving $(\mathsf{P}_j, m, \mathsf{CausalPast}(\mathsf{P}_j, m))$ on Causal_i wait until $\mathsf{CausalPast}(\mathsf{P}_j, m) \subseteq \mathsf{Delivered}(\mathsf{P}_i) \cup \{(\mathsf{P}_j, m)\}$. Then deliver (P_j, m) on Causal_i, and let $\mathsf{CausalPast}(\mathsf{P}_i) = \mathsf{CausalPast}(\mathsf{P}_i) \cup \mathsf{CausalPast}(\mathsf{P}_j, m)$. Also add (P_j, m) to $\mathsf{Delivered}(\mathsf{P}_i)$.

$\mathsf{FIFO2Causal}$: On $(\mathsf{P}_j, m, \mathsf{VectorClock}(\mathsf{P}_j, m))$ on FIFO_i at P_i, where $\mathsf{VectorClock}(\mathsf{P}_j, m) \leq \mathsf{Delivered}(\mathsf{P}_i) + \mathsf{P}_j$, deliver (P_j, m) on Causal_i and let $\mathsf{VectorClock}(\mathsf{P}_i) = \max\left(\mathsf{VectorClock}(\mathsf{P}_i), \mathsf{VectorClock}(\mathsf{P}_j, m)\right)$ and $\mathsf{Delivered}(\mathsf{P}_i) = \mathsf{Delivered}(\mathsf{P}_i) + \mathsf{P}_j$.

It should be clear when $\mathsf{FIFO2Causal}$ is run on top of a FIFO protocol, then it does exactly the same things as the inefficient protocol $\mathsf{Flood2Causal}$ and therefore is safe and live for exactly the same reasons. For instance, when using FIFO flooding, then the condition $\mathsf{CausalPast}(\mathsf{P}_j, m) \subseteq \mathsf{Delivered}(\mathsf{P}_i) \cup \{(\mathsf{P}_j, m)\}$ is true if and only if $\mathsf{VectorClock}(\mathsf{P}_j, m) \leq \mathsf{Delivered}(\mathsf{P}_i) + \mathsf{P}_j$. In this sense, vector clocks is just an efficient compressing of causal pasts on a FIFO network.

This argues the following result

Lemma 4.6 $\mathsf{FIFO2Causal} + \mathsf{FIFO} = \mathsf{Flood2Causal} + \mathsf{FIFO}$.

It is clear that

$$\mathsf{FIFO} \sqsubseteq \mathsf{Flood} \ .$$

The reason is that the behaviour of FIFO is a subset of the allowed behaviour of Flood. So it is safe to use FIFO whenever it is safe to use Flood. From $\mathsf{FIFO} \sqsubseteq \mathsf{Flood}$ we get that

$$\mathsf{Flood2Causal} + \mathsf{FIFO} \sqsubseteq \mathsf{Flood2Causal} + \mathsf{Flood} \ .$$

We have already proven that

$$\mathsf{Flood2Causal} + \mathsf{Flood} \sqsubseteq \mathsf{Causal} \ .$$

Putting these together we have that

$$\mathsf{FIFO2Causal} + \mathsf{FIFO} = \mathsf{Flood2Causal} + \mathsf{FIFO} \sqsubseteq \mathsf{Flood2Causal} + \mathsf{Flood} \sqsubseteq \mathsf{Causal} \ ,$$

which implies that

$$\mathsf{FIFO2Causal} + \mathsf{FIFO} \sqsubseteq \mathsf{Causal} \ .$$

We have argued the following result:

Theorem 4.7 $\mathsf{FIFO2Causal} + \mathsf{FIFO} \sqsubseteq \mathsf{Causal}$.

Notice that once all messages have been tagged with vector clocks, we can efficient check if there are in each others causal pasts or whether they are concurrent. It is easy to see that

$$(\mathsf{P}_i, m_i) \hookrightarrow (\mathsf{P}_j, m_j) \iff \mathsf{VectorClock}(\mathsf{P}_i, m_i) \leq \mathsf{VectorClock}(\mathsf{P}_j, m_j) \ .$$

This gives an efficient implementation of causality testing. We simply keep track of the vector clocks, and when we want to know whether $(\mathsf{P}_i, m_i) \hookrightarrow (\mathsf{P}_j, m_j)$ we simply check whether $\mathsf{VectorClock}(\mathsf{P}_i, m_i) \leq \mathsf{VectorClock}(\mathsf{P}_j, m_j)$.

When we have two message (P_i, m_i) and (P_j, m_j) for which $\mathsf{VectorClock}(\mathsf{P}_i, m_i) \not\leq \mathsf{VectorClock}(\mathsf{P}_j, m_j)$ and $\mathsf{VectorClock}(\mathsf{P}_j, m_j) \not\leq \mathsf{VectorClock}(\mathsf{P}_i, m_i)$, we say that the messages are concurrent. Neither of these messages have have affected each other. When two messages are concurrent we write

$$(\mathsf{P}_j, m_j) \| (\mathsf{P}_i, m_i) \ .$$

Exercise 4.4 (Safety and liveness of Vector Clocks) Above we said the vector-clock protocol needs to run on top of a FIFO flooding network. It turns out, however, that the protocol also works if run on a flooding network without FIFO. Prove that the protocol achieves causal delivery also when run on a network without FIFO. I.e., you are to prove that $\mathsf{FIFO2Causal} + \mathsf{Flood} \sqsubseteq \mathsf{Causal}$. The trick is to prove that FIFO2Causal is itself ensuring FIFO delivery.

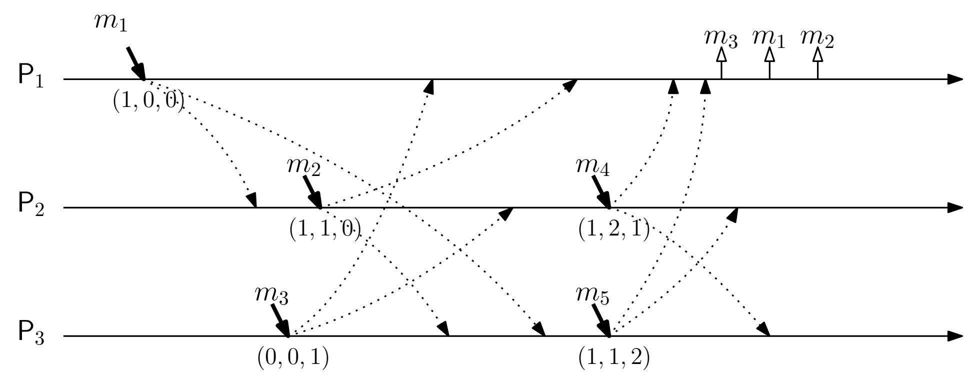


Figure 4.12 An illustration for total order. If we insist on causal order, then $\{m_1, m_2, m_5\}$ should be delivered in the order m_1, m_2, m_5 as $m_1 \hookrightarrow m_2 \hookrightarrow m_5$. And $\{m_2, m_4\}$ should be delivered in the order (m_2, m_4) as $m_2 \to m_4$. Similarly m_3 must be delivered before m_4 and m_5. Messages m_4 and m_5 are concurrent, and m_3 is concurrent with both m_1 and m_2. This gives a number of possible causal orders, for instance $(m_1, m_2, m_3, m_4, m_5)$ and $(m_2, m_1, m_3, m_5, m_4)$. But if we insist on total order, all parties have to do it in the same order. If we use the rule that we sort concurrent message on their vector clocks lexicographically, then m_3 should come first as $(0, 0, 1)$ is lexicographically less than $(1, 0, 0)$. And m_5 should come before m_4. So the total order should be $(m_3, m_1, m_2, m_5, m_4)$. Recall that a message m is deliverable when you received from all other parties a message with a vector clock larger than that of m. Therefore, when P_1 received m_4 and m_5 the messages m_1, m_2, m_3 become deliverable. They are delivered in the specified order. No other messages are deliverable at the current point in time. Do you see why?

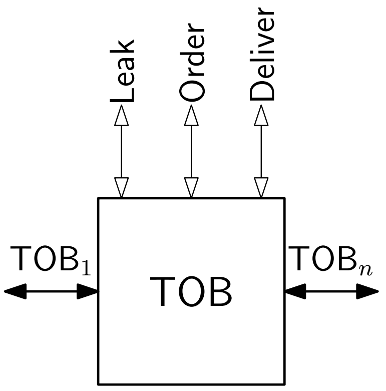
4.4 Total Order

This is the final consistency requirement we will look at.

Total Order If a correct P_k delivered (P_i, m) and then later delivered (P_j, m'), then it holds for all correct P_m that if they deliver (P_j, m'), then they earlier delivered (P_i, m).

Total order in itself does not imply FIFO or Causal Consistency, but can be combined with them. The reason is that, even if all players deliver received messages in the same order, it could still be that they are delivered in the wrong causal order, say, an answer is delivered before the question.

In Fig. 4.13 we give an ideal functionality TOB which has totally ordered delivery. We will now discuss how to implement TOB given Causal.



Ports The channel connects n parties named $\mathsf{P}_1, \ldots, \mathsf{P}_n$. For each party P_i it has a port called TOB_i. It has a special ports Leak and Deliver used to model that the network does not hide what is sent and that the adversary (or whomever controls Deliver) can determine when messages are delivered. It also has a port Order which can be used to determine the order in which messages are delivered. This order will be the same at all parties.

Init For each P_i it keeps a set $\mathsf{InTransit}_i$ which is initially empty. It also keeps a set Unordered.

Send On input (P_i, m) on Flood_i, it outputs (P_i, m) on Leak, and adds (P_i, m) to Unordered.

Order On input (P_i, m) on Order, where $(\mathsf{P}_i, m) \in \mathsf{Unordered}$, remove (P_i, m) from Unordered and add it at the back of the queues $\mathsf{InTransit}_j$ for $j = 1, \ldots, n$.

Deliver On input P_i on Deliver, where $\mathsf{InTransit}_i$ is non-empty, remove the head (P_k, m) from $\mathsf{InTransit}_i$ and output (P_k, m) in TOB_i.

Figure 4.13 TOB Network

When we require total order, we need to allow delayed local delivery, as two messages sent at the same time at different parties need to be delivered in the same order at all parties. So at least one of the parties have to wait with delivering its own message. As an illustration, consider Fig. 4.14.

Below we give an implementation which achieves both Causal Order and Total Order. It does so by first achieving Causal Order and then sorting the concurrent messages.

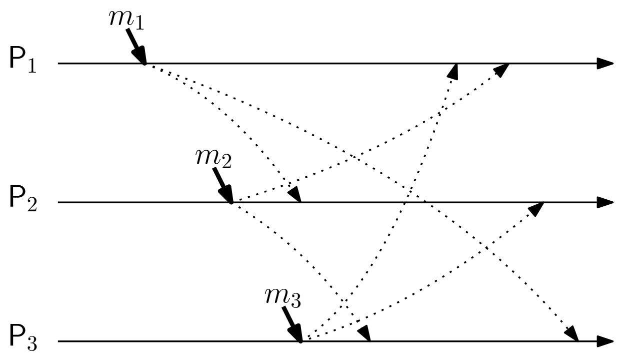


Figure 4.14 Three parties are sending three concurrent messages using Causal. They might arrive in different orders at different parties as they are concurrent. Assuming that Causal has immediate local delivery, they are delivered by Causal in the following ordering: $\mathsf{P}_1 : (m_1, m_3, m_2)$; $\mathsf{P}_2 : (m_2, m_1, m_3)$; $\mathsf{P}_3 : (m_3, m_2, m_1)$. But a protocol implementing TOB needs to deliver them in the same order. So two of the parties must wait with delivering their own message on TOB_i until receiving the message that must be output first. If we order concurrent messages lexicographically on their vector clocks, then the correct total order would be (m_3, m_2, m_1). So P_1 can for instance not deliver m_1 locally before having delivered m_3.

4.4.1 An Implementation Assuming we can Detect Safe Messages

We start with a flooding network Causal with Causal Order. Note then that only concurrent messages can get delivered in different order at different parties, as $(\mathsf{P}_i, m_i) \hookrightarrow (\mathsf{P}_j, m_j)$ implies that (P_i, m_i) always gets delivered before (P_j, m_j). Also, we can output the concurrent messages in any order we want without risking to break Causal Order. For the concurrent messages we can then just let all parties sort their concurrent messages using any deterministic total ordering.

As an example, they could sort all messages by lexicographic order of the vector clock, so a vector clock $(0, 0, 1)$ would be smaller than $(1, 0, 0)$. We write this as $\mathsf{VectorClock}(\mathsf{P}_i, m_i) < \mathsf{VectorClock}(\mathsf{P}_j, m_j)$. For brevity we just write $(\mathsf{P}_i, m_i) < (\mathsf{P}_j, m_j)$ instead of $\mathsf{VectorClock}(\mathsf{P}_i, m_i) < \mathsf{VectorClock}(\mathsf{P}_j, m_j)$. Given any causal order $\hookrightarrow$ on messages we can then extend it into a total+causal order $\rightharpoonup$ as follows. We say that

$$(\mathsf{P}_i, m_i) \rightharpoonup (\mathsf{P}_j, m_j)$$

if and only if

$$((\mathsf{P}_i, m_i) \hookrightarrow (\mathsf{P}_j, m_j)) \vee ((\mathsf{P}_i, m_i) \| (\mathsf{P}_j, m_j) \wedge (\mathsf{P}_i, m_i) < (\mathsf{P}_j, m_j)) \ .$$

Recall that $(\mathsf{P}_i, m_i) \| (\mathsf{P}_j, m_j)$ means that the messages are concurrent, i.e., $(\mathsf{P}_i, m_i) \not\hookrightarrow (\mathsf{P}_j, m_j)$ and $(\mathsf{P}_j, m_j) \not\hookrightarrow (\mathsf{P}_i, m_i)$. So, in words we say that $(\mathsf{P}_i, m_i) \rightharpoonup (\mathsf{P}_j, m_j)$ if either the messages are causally related or they are concurrent and (P_i, m_i) is lexicographically smaller than (P_j, m_j).

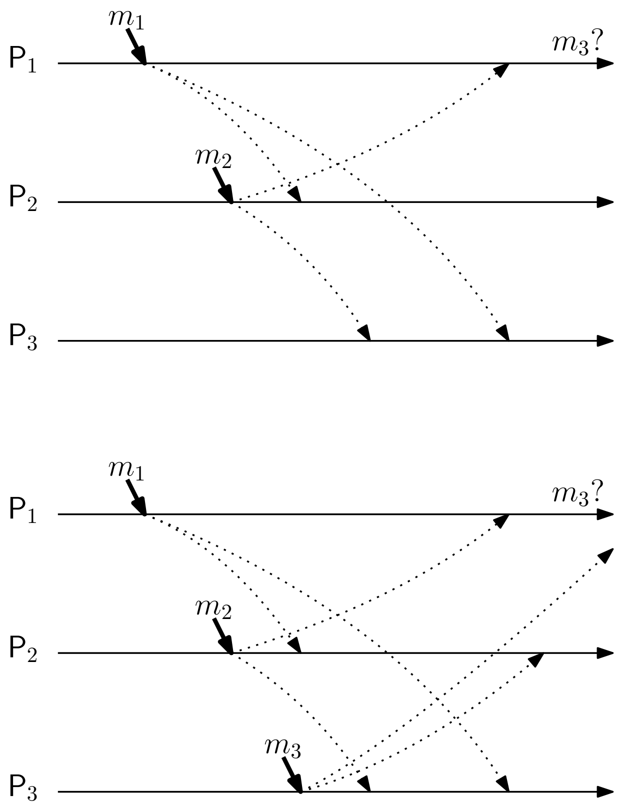
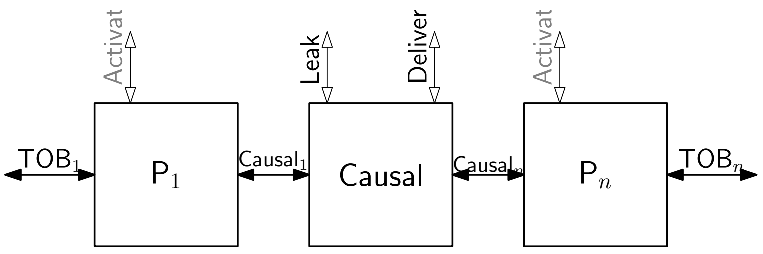


Figure 4.15 Two executions similar to Fig. 4.14, but with P_3 not sending a message in the top execution, and with P_3's message to P_1 still being in transit in the bottom execution. In the top execution the correct total order is (m_3, m_2) as we sort lexicographically on vector clocks for concurrent messages and $(0,0,1) < (1,0,0)$. In the bottom configuration the correct total order is (m_3, m_2, m_1). Now consider the two executions from the point of view of P_1. It receives the same messages at the same times. So it cannot know whether it is in the top execution or the bottom execution. Therefore it cannot deliver any message in either setting. In the bottom setting it did not get m_3 yet. And in the top setting it does not know whether there might be an m_3 on its way from P_3. If no more messages are ever sent in the system, this situation would go on forever. At any point in time it could be that P_3 just sent the message m_3 and that P_1 therefore should wait a bit longer. So the system is in an apparent deadlock. Any totally-ordered broadcast running on top of a Causal network resource needs to find a way to rule out whether there are still concurrent messages on their way. When we can be sure for a message m that no message concurrent with m can ever arrive, we call m safe. In the above executions there are no safe messages.

Our goal is now to construct a protocol which delivers according to $\rightharpoonup$. There is one big problem to be solved to achieve this. It is hard to know in a system



All parties P_i run the following activation rules

Init Initialise a set $\mathsf{Unordered}_i$, initially empty.
Send On input (P_i, m) on TOB_i, send (P_i, m) on Causal_i.
Delay On (P_j, m) on Causal_i, add (P_j, m) to $\mathsf{Unordered}_i$.
Deliver If it holds for the smallest element (P_j, m_j) in $\mathsf{Unordered}_i$ (ordered according to $\rightharpoonup$) that $\text{IsSafe}(\mathsf{P}_j, m_j) = \top$, then remove (P_j, m_j) from $\mathsf{Unordered}_i$ and output (P_j, m_j) on TOB_i.

Figure 4.16 The Protocol Causal2TOB.

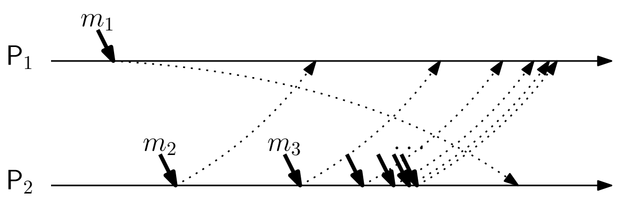


Figure 4.17 A problematic case for Causal2TOB. Since we order concurrent messages lexicographically it could happen that P_2 sends an infinite number of messages before m_1 arrives at P_2. Since we sort concurrent messages lexicographically on their vector clocks, these messages are all less than m_1 in $\rightharpoonup$, so m_1 is never delivered. This is why we made the light assumption on Causal that it does not allow one party to send infinitely many messages in finite time.

communicating via Causal whether more concurrent messages are on their way. For an illustration of this, consider Fig. 4.15. We need to come up with a way such that we for all messages (P_i, m_i) eventually can determine that no message which is concurrent with (P_i, m_i) is still on its way.

We first assume that we can solve this problem and show that then we are done. We are going to assume that we have a procedure IsSafe that the parties can run locally. It takes as input a message (P_j, m_j) that was sent via Causal and it outputs $\top$ or $\bot$. When it holds that $\text{IsSafe}(\mathsf{P}_j, m_j) = \top$ (at P_i), then we call (P_j, m_j) safe (at P_i). The procedure IsSafe should have a safety and a liveness property:

Safety If at some point in time t_1 it holds that IsSafe$(\mathsf{P}_j, m_j) = \top$ at party P_i and it holds at some later point in time t_2 that P_i receives (P_k, m_k) on Causal_i, then $(\mathsf{P}_j, m_j) \hookrightarrow (\mathsf{P}_k, m_k)$.

Liveness It holds for all P_i and all messages (P_j, m_j) received at P_i from Causal that eventually IsSafe$(\mathsf{P}_j, m_j) = \top$ at party P_i. And when it happens at some point in time that IsSafe$(\mathsf{P}_j, m_j) = \top$, then it holds for all future times too.

The liveness property just says that eventually we will consider all messages safe. The safety property says that once (P_j, m_j) is considered safe, then all later messages to arrive will be in the causal future of (P_j, m_j). This is also the minimal property making IsSafe useful. What we want is that there is no message concurrent to (P_j, m_j) on its way. This is the same as saying that for all later messages it should hold that $(\mathsf{P}_j, m_j) \not\parallel (\mathsf{P}_k, m_k)$. This is the same as requiring that $(\mathsf{P}_j, m_j) \hookrightarrow (\mathsf{P}_k, m_k)$ or $(\mathsf{P}_k, m_k) \hookrightarrow (\mathsf{P}_j, m_j)$. But since (P_k, m_k) arrives later than (P_j, m_j) it cannot be the case that $(\mathsf{P}_k, m_k) \hookrightarrow (\mathsf{P}_j, m_j)$, as Causal promises to deliver in the order $\hookrightarrow$. So it must be the case that $(\mathsf{P}_j, m_j) \hookrightarrow (\mathsf{P}_k, m_k)$.

If we have a safe and live procedure IsSafe, then we can implement TOB using the protocol $\mathsf{Causal2TOB}$ in Fig. 4.16.

Theorem 4.8 *Assuming that* IsSafe *is live and safe we have that* $\mathsf{Causal2TOB} + \mathsf{Causal} \sqsubseteq \mathsf{TOB}$.

Proof We first argue safety. Assume that some (P_j, m_j) is output on TOB_i by P_i. We want to argue that if $(\mathsf{P}_k, m_k) \rightharpoonup (\mathsf{P}_j, m_j)$, then (P_k, m_k) was already output. When (P_j, m_j) was output it was the case that IsSafe$(\mathsf{P}_j, m_j) = \top$ by construction of the protocol. It follows from safety of IsSafe and $(\mathsf{P}_k, m_k) \rightharpoonup (\mathsf{P}_j, m_j)$ that when IsSafe$(\mathsf{P}_j, m_j) = \top$ then (P_k, m_k) already arrived at P_i. To see this note that when $(\mathsf{P}_k, m_k) \rightharpoonup (\mathsf{P}_j, m_j)$ then $(\mathsf{P}_k, m_k) \hookrightarrow (\mathsf{P}_j, m_j)$ or $(\mathsf{P}_k, m_k) \parallel (\mathsf{P}_j, m_j)$. If $(\mathsf{P}_k, m_k) \hookrightarrow (\mathsf{P}_j, m_j)$, then (P_k, m_k) arrived because we communicate via Causal. If $(\mathsf{P}_k, m_k) \parallel (\mathsf{P}_j, m_j)$, then (P_k, m_k) arrived because IsSafe$(\mathsf{P}_j, m_j) = \top$ and IsSafe is safe. So, we can conclude that (P_k, m_k) arrived. This ensures that (P_k, m_k) is either in $\mathsf{Unordered}_i$ or already output. If it was already output we are done, as this is what we should ensure. And it cannot be the case that it is in $\mathsf{Unordered}_i$ and at the same time that $(\mathsf{P}_k, m_k) \rightharpoonup (\mathsf{P}_j, m_j)$, as we pick (P_j, m_j) to be the smallest element in $\mathsf{Unordered}_i$ according to $\rightharpoonup$. This concludes the proof of safety.

We then argue liveness. We would like to argue that eventually all messages get delivered on TOB_i. At any point of time in an execution consider a message (P_j, m_j) which has been sent. We argue that it eventually gets delivered on TOB_i. At the point in time where it is delivered at all parties on Causal_i at most a finite number of elements were sent. Call the set of these messages X. Note, this is the set of all messages ever sent up to the point where (P_j, m_j) arrived at all parties. Let Y be the set of messages sent on Causal which are not in X. Note that all message in Y are in the causal future of (P_j, m_j). Therefore, if $(\mathsf{P}_k, m_k) \rightharpoonup (P_j, m_j)$, then $(\mathsf{P}_k, m_k) \in X$.

For all elements in X it holds that they eventually arrive. Since there are finitely many of them, they will all eventually arrive. And then it will also eventually happen that IsSafe$(\mathsf{P}_k, m_k) = \top$ for all of them. At this point one of them, (P_k, m_k) will be the smallest message in $\mathsf{Unordered}_i$. Since IsSafe$(\mathsf{P}_k, m_k) = \top$ it will get delivered. Then the second smallest get delivered and so on. Since there are finitely many messages smaller then (P_j, m_j) it will eventually be the turn of (P_j, m_j) to be delivered. □

4.4.2 Detecting Safety

We now discuss how to implement IsSafe. Recall that we for a given message (P_j, m_j) need to detect when all messages (P_k, m_k) which could be concurrent with (P_j, m_j) have arrived at P_i. Here are three ways to do it.

Timeouts

Assume that we cheat and give each party a clock and we assume that we know an upper bound Δ on the network delivery time. When P_i receives (P_j, m_j) at time t it then knows that all other parties receive (P_j, m_j) before time $t + \Delta$. So all messages sent after time $t + \Delta$ are in the casual future of (P_j, m_j) as they were sent after (P_j, m_j) arrived. So a concurrent message (P_k, m_k) was sent before time $t + \Delta$. So it will arrive at party P_i before time $t + 2\Delta$. So, IsSafe will simply annonce (P_j, m_j) safe if it arrived at least 2Δ second ago. One disadvantage of this method is that you need to know a worst case upper bound on network delivery time. Another disadvantage is that it adds a waiting time on delivering messages which is twice the worst-case network delivery time. This could make $\mathsf{Causal2TOB}$ very slow.

Common Causal-Past

We now consider a fully asynchronous method. Let (P, m) be a message received by P_i. We call (P, m) $\mathsf{safe\ relative}$ to P_j if P_i received a message (P_j, m_j) such that $(\mathsf{P}, m) \hookrightarrow (\mathsf{P}_j, m_j)$. We let IsSafe$(\mathsf{P}, m) = \top$ when (P, m) is safe relative to P_j for all P_j.

We now argue that the above IsSafe is safe. We have to argue that when IsSafe$(\mathsf{P}, m) = \top$ then no concurrent messages are on their way. It is enough to argue that it holds for all P_j that there are no concurrent messages on their way from P_j. When IsSafe$(\mathsf{P}, m) = \top$ then we know that (P, m) is safe relative to all P_j. So it is enough to argue that if (P, m) is safe relative to all P_j then there is no concurrent messages sent by P_j on its way.

So assume that (P, m) is safe relative to all P_j. This means P_i received a message (P_j, m_j) such that $(\mathsf{P}, m) \hookrightarrow (\mathsf{P}_j, m_j)$. We break the set of messages sent by P_j into two. The set B of messages sent before sending (P_j, m_j) and the set A of messages sent after sending (P_j, m_j). For a message $(\mathsf{P}_j, m') \in A$ we know that $(\mathsf{P}_j, m_j) \to (\mathsf{P}_j, m')$. We get from transitivity and $(\mathsf{P}, m) \hookrightarrow (\mathsf{P}_j, m_j)$ that $(\mathsf{P}, m) \hookrightarrow (\mathsf{P}_j, m')$. Therefore $(\mathsf{P}, m) \not\parallel (\mathsf{P}_j, m')$.

For a message $(\mathsf{P}_j, m') \in B$ we know that $(\mathsf{P}_j, m') \to (\mathsf{P}_j, m_j)$. Therefore P_i

received (P_j, m') before receiving (P_j, m_j) (we use Causal as network resource)). Therefore (P_j, m') is no longer on its way to P_i.

So it holds for all messages sent from P_j that either they are not concurrent to (P_j, m), or they are already delivered to P_i. In other words, all concurrent messages were already delivered. This is what we had to show.

This method also has liveness if it is guaranteed that all parties always sends a message again at some point in time. Consider for illustration a very busy TOB where all parties are sending messages all the time, say they send a message at least every ϵ seconds. Now the time for a message (P_j, m_j) to become safe is twice the actual network delay, which could be much smaller than twice the worst-case network delay. To see this consider the time t where P_j sends (P_j, m_j). Let Δ' be the currently actual network delay. Within $t + \Delta'$ the message (P_j, m_j) is received by all parties. Since they send messages all the time, they will all send a message a time $t + \Delta' + \epsilon$. These messages are all in the causal future of (P_j, m_j) and they arrive at all other parties before time $t + \epsilon + 2\Delta'$. At this point (P_j, m_j) is safe at all parties.

Flushing to Ensure Liveness

We now add liveness to the above IsSafe without assuming that all parties always send a message again. We can cheat a bit less with the time, and assume that each party has a clock which is an integer and which is always progressing. This will allow the parties to do something "now and then". By "now and then" we just mean that a party always will do it again. For instance each time the clock progressed one click. Now and then each party P_i will then send out a message $(\textsc{Flush}, \mathsf{P}_i, f_i)$, where f_i is the clock. This establishes the above situation, where all parties are always sending messages. Hence IsSafe is live and safe.

Exercise 4.5 (Lamport Clocks) It turns out that if vector clocks are used for creating total order, then one can use a simpler version where only one integer is sent along with each message. Each party holds a counter c_i, initially 0. It is incremented every time a message is sent. It is sent along with the new message. When a message is received the receiver sets its counter to be $\max(c_i, c(\mathsf{P}, m))$, where $c(\mathsf{P}, m)$ is the counter of the message. Now sort all messages lexicographically first on $c(\mathsf{P}, m)$ and then on who sent the message as tie breaker. Then deliver in that order. Work out the details and prove that it works. The trick is to see that $(\mathsf{P}_i, m_i) \hookrightarrow (\mathsf{P}_j, m_j) \Rightarrow c(\mathsf{P}_i, m_i) \le c(\mathsf{P}_j, m_j)$.

Exercise 4.6 (Implement a Simple Peer-to-Peer Ledger) Modify your code from Exercise 2.4 to add the following features:

1. The system now no longer broadcasts strings and prints them. Instead it implements a distributed ledger. Each client keeps a `Ledger` (see Fig. 4.18).
2. Each client can make `Transactions`. When they do all other peers eventually update their ledger with the transaction.

```
package account

import ( "sync" )

type Ledger struct {
     Accounts map[string]int
     lock sync.Mutex
}

func MakeLedger() *Ledger {
     ledger := new(Ledger)
     ledger.Accounts = make(map[string]int)
     return ledger
}
```

Figure 4.18 The `Ledger` type.

```
package account

type Transaction struct {
     ID string
     From string
     To string
     Amount int
}

func (l *Ledger) Transaction(t *Transaction) {
     l.lock.Lock() ; defer l.lock.Unlock()

     l.Accounts[t.From] -= t.Amount
     l.Accounts[t.To] += t.Amount
}
```

Figure 4.19 The `Transaction` type.

3. The system should ensure eventual consistency, i.e., if all clients stop sending transactions, then all ledgers will eventually be in the same correct state.
4. Your system only has to work if there are two phases: first all the peers connect, then they make transactions. But if you want to accommodate for later comers a way to do it is to let each client keep a list of all the transactions it saw and then forward them to clients that log in late.

Implement as follows:

1. Keep a sorted list of peers.
2. When connecting to a peer, ask for its list of peers.
3. Then add yourself to your own list.
4. Then connect to the ten peers after you on the list (with wrap around).
5. Then broadcast your own presence.

6. When a new presence is broadcast, add it to your list of peers.
7. When a transaction is made, broadcast the `Transaction` object.
8. When a transaction is received, update the local `Ledger` object.

Add this to your report:

1. Test you system and describe how you tested it.
2. Discuss whether connection to the next ten peers is a good strategy with respect to connectivity. In particular, if the network has 1000 peers, how many connections need to break to partition the network?
3. Argue that your system has eventual consistency if all processes are correct and the system is run in two-phase mode.
4. Assume we made the following change to the system: When a transaction arrives, it is rejected if the sending account goes below 0. Does your system still have eventual consistency? Why or why not?

Exercise 4.7 (better total-order) Look at Fig. 4.12 again. Neither P_2 nor P_3 can deliver m_3. This sort of makes sense. If P_1 had sent a new message right after sending m_1 it would have vector clock $(2, 0, 0)$ and be concurrent with m_3. So depending on how we sort concurrent messages this message might have to be delivered before m_3. However, we have decided to use a specific way to order concurrent messages, namely the lexicographic order with the sender identity being the most significant "digit" and P_1 being the first to be considered. For this exercise we change the convention and say that higher numbered parties are considered first. So if P_1 and P_3 sends messages concurrently with vector clocks $(1, 0, 0)$ and $(0, 0, 1)$, then we deliver $(0, 0, 1)$ With this knowledge we can do better. So, a message with vector clock $(2, 0, 0)$ would come after m_3. In fact, it is easy to see that with the way we sort, the vector clock $(0, 0, 1)$ of m_3 is the smallest possible vector clock of a message according to $\rightharpoonup$, so all parties could deliver m_3 immediately when receiving it! This proposes the following more efficiently rule for delivering message: A message m becomes deliverable when it follows from the vector clocks of the messages seen that no message with a smaller vector clock according to $\rightharpoonup$ could be on its way to you. Deliver all deliverable messages in the order $\rightharpoonup$. With this new and better rule, when would the various messages get delivered in Fig. 4.12, and which messages still cannot be delivered.

5

Confidentiality (DRAFT)

Contents

5.1 Confidentiality, Secret-Key Systems

A secret-key cryptosystem consists of three algorithms G, E, D, one for generating a key, one for encryption and one for decryption. The algorithm G takes as input a key length κ and usually produces a key by just outputting a randomly chosen bit string of that length. Usually only some fixed key lengths are allowed, like 128, 242 and 256. The algorithm E takes as input a key k and a message (or 'plaintext') m and outputs a *ciphertext* c. The algorithm D takes a ciphertext c and a key k and produces a plaintext $D_K(c)$. The system must be such that if the *same* key is used for encryption and decryption, the original plaintext is recovered. In other words, it must hold that:

$$m = D_K(E_K(m)).$$

In addition, we would like the system to be *secure*, in the sense that an adversary who sees c but does not know k should have no idea whatsoever about which plaintext c represents.

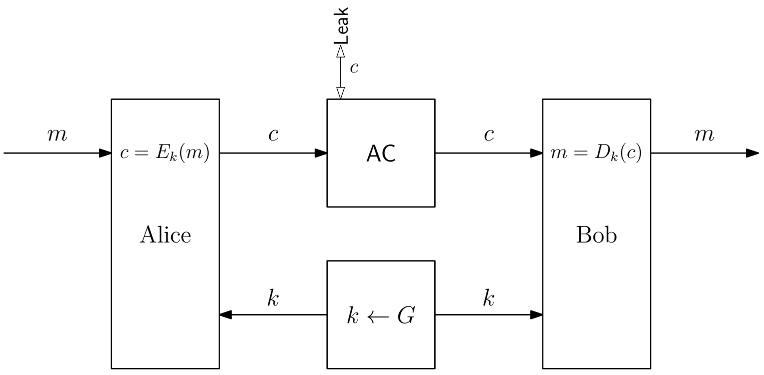


Figure 5.1 An illustration of how a symmetric encryption scheme (G, E, D) is used to send a single message. Alice has to send a message m to Bob over an authenticated channel which might leak the transmitted data. Somehow they managed to run the key generator G in the past and learn a common key k. When m arrives, Alice can compute and send $c = E_K(m)$. This ciphertext is allowed to leak. When receiving c, Bob can compute $m = D_K(c)$ and output m. It is essential that k is known only by Alice and Bob. We call the protocol $\Pi_{G,E,D}$.

5.1.1 The One-time Pad and Perfect Secrecy

If each encryption key is only used once, it is not too difficult to design an unbreakable encryption algorithm, which takes the form of the so-called *one-time pad*, constructed as follows. Assuming the message is a string of bits labeled $m_1, ..., m_t$, the key will be a random bit-string of the same length labeled $k_1, ..., k_t$. We encrypt by taking the bit-wise xor of the strings, i.e., the ciphertext $c_1, ..., c_t$ is defined by $c_i = m_i \oplus k_i$. The receiver, who also knows the same key, can recover the plaintext by decrypting each i'th bit as:

$$c_i \oplus k_i = (m_i \oplus k_i) \oplus k_i = m_i \oplus (k_i \oplus k_i) = m_i.$$

The intuition for security is as follows: given $c_1, ..., c_t$, since any key string $k_1, ..., k_t$ is possible, $c_1, ...c_t$ could decrypt to any plaintext whatsoever. So the eavesdropper learns nothing about the message from seeing the ciphertext. A more precise result is easy to show:

Theorem 5.1 *When* one-time pad *is used for encryption, the ciphertext is always a uniformly distributed bit string, in particular, it is independent of the plaintext.*

Proof For illustration, we first show the simple case where we encrypt just one bit. Since the key bit is chosen independently of the message, we calculate as follows:

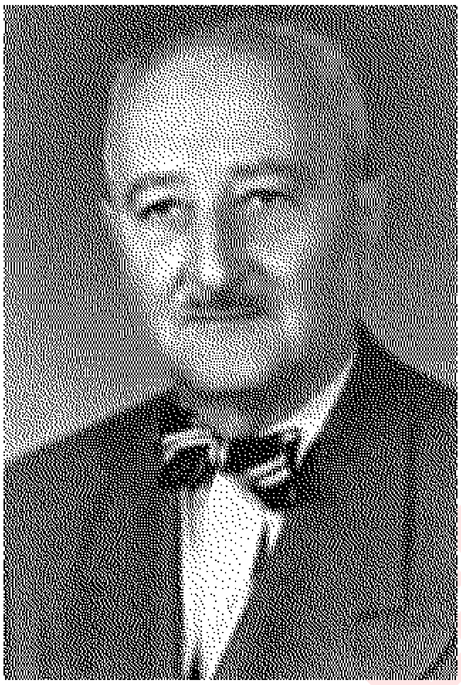


Figure 5.2 William Frederick Friedman (September 24, 1891–November 12, 1969) was a US Army cryptographer who ran the research division of the Army's Signal Intelligence Service (SIS) in the 1930s. Friedman invented many important cryptanalytic techniques, including the index of coincidence, and coined the term *cryptoanalysis*. During WWII SIS broke Japan's PURPLE cipher, thus disclosing Japanese diplomatic secrets before America's entrance into World War II. The picture is of poor quality as it is monochrome: each pixel is either completely black or completely white.

$$\begin{aligned}\Pr[c=0] &= \Pr[m=0,k=0] + \Pr[m=1,k=1] \\ &= \Pr[m=0]\Pr[k=0] + \Pr[m=1]\Pr[k=1] \\ &= \Pr[m=0]\cdot\frac{1}{2} + \Pr[m=1]\cdot\frac{1}{2} \\ &= \frac{1}{2}(\Pr[m=0]+\Pr[m=1]) \\ &= \frac{1}{2}\end{aligned}$$

In the general case, we also have to use that the key bits are chosen independently of each other, but otherwise the basic idea is the same: we can show that for any message length t, each ciphertext occurs with probability 2^{-t}. For simplicity, we do this only for the all-0 ciphertext of length t: $00\ldots0 = 0^t$. Note

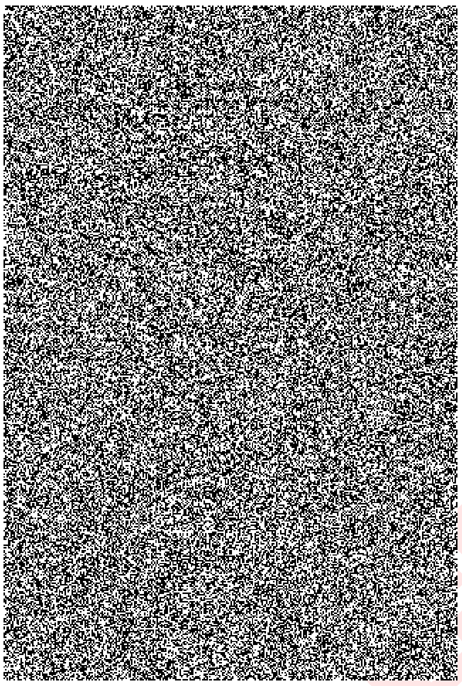


Figure 5.3 This is a one-time pad encryption of the monochrome picture of Friedman in Fig. 5.2. First the picture was converted to a bit string. A white pixel has been represented by a 1-bit and a black pixel has been represented by a 0-bit. Then the bit string was encrypted using one-time pad with a key which is as long as the picture file. This flips each bit in the file with probability 50% independently of the other bits, so after this all the bits are uniformly random and independent. Then the bit string was converted back into a picture with the same rule as above: 0 becomes black and 1 becomes white. Since all the bits are uniformly random and independent the picture is just pure noise.

that for the ciphertext to be 0, the plaintext and key must take the same value, but this can be any value x.

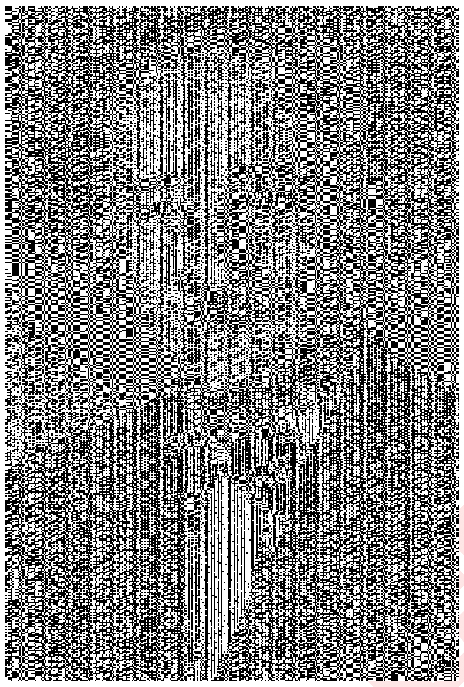


Figure 5.4 This is a failed attempt at a one-time pad encryption of the monochrome picture of Friedman in Fig. 5.2. First the picture was converted to a bit string using the same rule as in Fig. 5.3. Then the bit string was encrypted using one-time pad. But a mistake was made: the key used is only 16 bits long, and it was used repeatedly. This means that each 16-bit block in the picture file was encrypted using the same key. In particular, each each bit in the file flips with probability 50%, but two bits which are 16 bits apart are encrypted using the same key, so they are both are flipped or not. Then the bit string was converted back into a picture with the same rule as above: 0 becomes black and 1 becomes white. Since the picture is a multiple of 16 bits wide, each column is encrypted using the same bit. So either the column is left as it was, or all the pixels are flipped. Your eyes are enough to "cryptanalyse" this failed attempt at a one-time pad encryption. It makes it clear that we need a key which is as long as the picture when we use one-time pad.

$$\begin{aligned}
\Pr[c = 0^t] &= \sum_{x\in\{0,1\}^t} \Pr[m = x, k = x] \\
&= \sum_{x\in\{0,1\}^t} \Pr[m = x] \Pr[k = x] \\
&= \sum_{x\in\{0,1\}^t} \Pr[m = x]\, 2^{-t} \\
&= 2^{-t} \sum_{x\in\{0,1\}^t} \Pr[m = x] \\
&= 2^{-t}
\end{aligned}$$

□

What this shows is that the ciphertext has the same distribution *no matter what the plaintext was*, which of course means that there is no way to learn anything about the plaintext from seeing the ciphertext. We say that a cryptosystem has *perfect secrecy* when the ciphertext is independent of the plaintext. Note that perfect secrecy holds no matter how much computing power the eavesdropper has – in other words, the one-time pad is unconditionally secure.

However, the argument above works only if each key string is used *once*, otherwise the key bits would not be independent: some of the key bits used at later positions would be copies of bits used before. So this means that we need as many random key bits as we have message bits, and so the one-time pad is quite useless in practice. It is not hard to see that this is no coincidence: no matter how smart we try to be, there is no way around this problem:

Theorem 5.2 *Suppose a cryptosystem can handle $\mathcal{M}$ possible plaintexts, and uses $\mathcal{C}$ ciphertexts and $\mathcal{K}$ keys. If the system has perfect secrecy, then it must be the case that $\mathcal{K} \geq \mathcal{C} \geq \mathcal{M}$.*

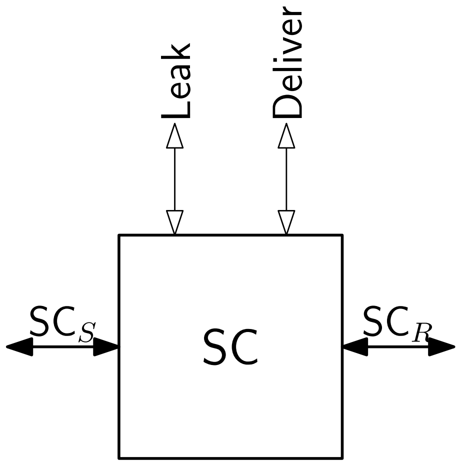
Proof For any cryptosystem it must hold that $\mathcal{C} \geq \mathcal{M}$, otherwise encryption would have to send different plaintexts to the same ciphertext, and we could not decrypt correctly. Now, consider some plaintext m. Note that for every ciphertext c, there must exist some key K_c for which $E_{K_c}(m) = c$. If this was not the case, there would be some ciphertext c_0 that could not occur when the plaintext is m. Then we would not have perfect secrecy: if the adversary sees that ciphertext c_0 was sent, he could conclude that the plaintext was not m. So we conclude that there is at least one key for every ciphertext, i.e., $\mathcal{K} \geq \mathcal{C}$. □

To see what this result means, note that if we did try to use the same "one-time" pad key twice, then the number of possible keys would be *smaller* than the number of possible messages, and then above result says that perfect secrecy is not possible.

5.1.2 Securely Implementing a Secure Channel

We now relate the notion of a secure symmetric encryption scheme with the notion of securely implementing a secure channel. We use the network resource SC in Fig. 5.5 to model a secure channel. Notice that as opposed to AC the network resource SC does *not* leak the message m. That is what makes it a model of a secure channel. It does, however, leak the length of m. This is because we cannot expect an implementation to completely hide the length of the message being sent. An encryption of a single bit is hopefully shorter than an encryption of a two hour movie.

Recall now that the canonical way we envision to use a symmetric encryption scheme is as in Fig. 5.1. We could ask the question: when it holds for the protocol $\Pi_{G,E,D}$ in Fig. 5.1 that it securely implements SC, i.e., when is it the case that $\Pi_{G,E,D} \sqsubseteq \mathsf{SC}$? Recall that to prove $\Pi_{G,E,D} \sqsubseteq \mathsf{SC}$ we need to come up with a



Ports The channel connects two parties S and R. For each party X it has a user port called SC_X. It has a special port called Leak, which is used to model that the channel does not hide the *length* of the message being sent. It has a special port Deliver which is used to model that it is its environment which controls when messages are delivered.

Init It keeps a set InTransit which is initially empty. It initializes a counter $\mathsf{mid} = 0$.

Send On input m on SC_S, where mid is a fresh message identifier, it outputs $(\mathsf{mid}, |m|)$ on Leak, adds (mid, m) to InTransit, and lets $\mathsf{mid} = \mathsf{mid} + 1$..

Deliver On input mid on Deliver, where there exist $(\mathsf{mid}, m) \in \mathsf{InTransit}$, it removes (mid, m) from InTransit and outputs (mid, m) on SC_R.

Figure 5.5 The SC network resource modeling a secure channel.

simulator Sim such that $\Pi_{G,E,D} \equiv \mathsf{SC} + \mathsf{Sim}$. Recall that when a message m_1 is sent in $\Pi_{G,E,D}$ the channel AC leaks $E_K(m_1)$. When a message m_1 is sent in SC + Sim, then SC leaks only $|m_1|$. Fig. 5.6 shows $\Pi_{G,E,D}$ and SC + Sim in juxtaposition. So, the job of Sim is to get as input $|m_1|$ and then output $E_K(m_1)$. This might seem impossible at first: the simulator is given only the length of m_1, so it cannot compute m_1 itself, as there are many messages with the same length. But recall that when k is unknown and (G, E, D) is the one-time pad, then $E_K(m_1)$ and $E_K(m_2)$ have the exact same distributions for all m_1 and m_2 of the same length. Therefore Sim can just output $E_K(m_2)$ for some m_2 with $|m_2| = |m_1|$. The simulator in Fig. 5.6 picks a uniformly random such message m_2, but any message of the right length will do.

5.1.3 Practical systems, Definition of Security

In real life, it is clear that we will have to use the same key for many different messages, or at least for a message much longer than the key. This means in practical systems we typically have to settle for computational security, i.e., we

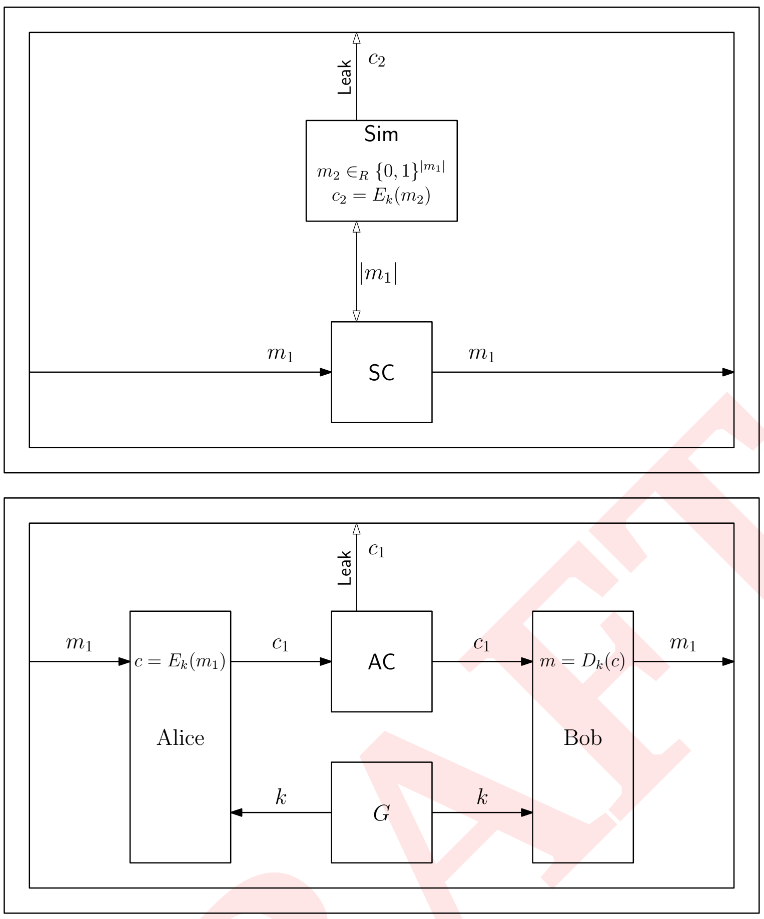


Figure 5.6 $\Pi_{G,E,D}$ and $\mathsf{SC} + \mathsf{Sim}$ in juxtaposition. The simulator outputs a uniformly random message m_2 of the same length as m_1. This ensures that $\Pi_{G,E,D} \equiv \mathsf{SC} + \mathsf{Sim}$ when (G, E, D) is the one-time pad and only one message m_1 is sent and $|k| = |k|$.

build systems that we hope the adversary cannot break because he would have to spend an unrealistically large amount of time to do so.

This has several consequences for how we should design cryptosystems and use them: First, the adversary should have no idea what we are sending even if we use the system several times, so if for instance we send the same message twice, an eavesdropper should not be able to tell that this was what happened. If the encryption algorithm really only takes key and message as inputs, then if we send the same message twice, two identical ciphertexts will be produced, and this will be visible to an eavesdropper.

Therefore, good encryption algorithms work not only with the input k, m as

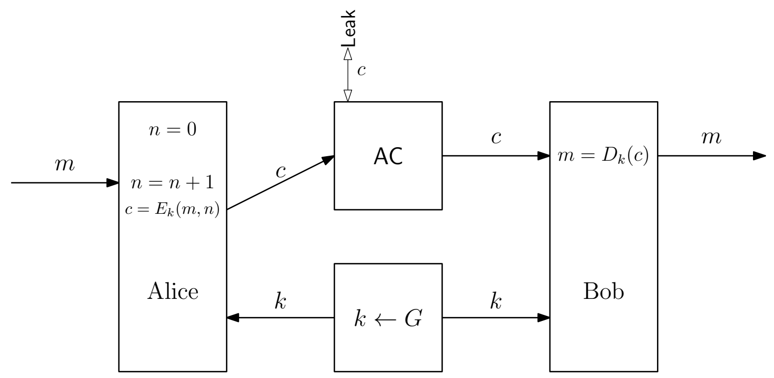


Figure 5.7 An illustration of how a symmetric encryption scheme (G, E, D) is used for sending multiple messages. It works as in Fig. 5.1, but each encryption also uses a nonce, here a counter. Initially the counter is set to $n = 0$. Each time a new message is sent, it is incremented and given to the encryption algorithm E as input.

input, they also make a choice of some variable whose value changes from one encryption operation to the next. Such a variable is often called a *nonce* ("number used once"). It must be chosen in some way that guarantees that we will not use the same value twice. A nonce can take the form of a counter or it may consist of random bits that are freshly chosen every time we encrypt something.

As a result, if we encrypt the same message twice, the value of the nonce n will be different in the two cases, and so (if we design the encryption algorithm properly) this gives us a chance to make sure that even two encryptions of the same message look like unrelated, random strings of bits. When we want to emphasize that a nonce n was used in the encryption process we write

$$c = E_K(m, n),$$

where it must of course still be the case that $m = D_K(E_K(m, n))$. We emphasize, however, that if the particular value of n is not important, we will sometimes just write $E_K(m)$ even if the encryption algorithm is defined to choose a nonce internally.

Here is an informal version of the definition of security one usually wants from cryptosystems with computational security: Consider an adversary who plays the following game: he sends a message m to an oracle O who has a secret key k inside. (The oracle is just a "black-box" that the adversary can play with, and it is a convenient abstract way to model what kind of access the adversary has to our system). The oracle sends back a ciphertext c, which is computed in one of two ways: In case 1) $c = E_K(m, n)$, in case 2) $c = E_K(r, n)$ where r is random message of the same length as m. In both cases a different nonce n is used for each encryption. The adversary may send as many messages as he likes but the oracle stays in case 1) or 2) all the time. The security defintion can then be stated as follows.

Definition 5.3 Consider any adversary who plays the above game and whose computing power is limited in the sense that whatever algorithm he runs terminates in time much less than the time it takes to try all possible keys in the cryptosystem (see below for a detailed discussion on exhaustive key search). No such adversary can guess whether he is in case 1) or 2) (with probability better than essentially a random guess).

What this definition is saying is that in real life when an adversary looks at encrypted data, even if he knows (or can even to some extent control) the data that is encrypted, for all he knows, the encrypted data may as well contain unrelated random garbage. So in a very strong sense, the adversary has no idea about what is encrypted.

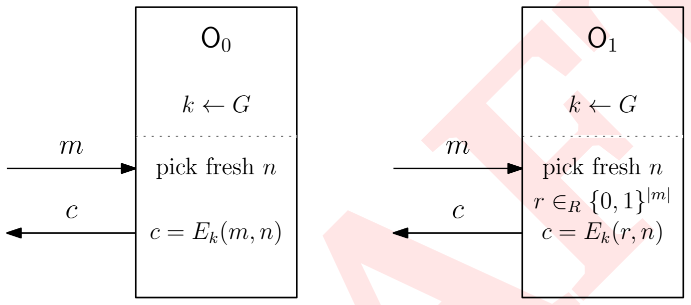


Figure 5.8 An illustration of the security notion indistinguishability under chosen-plaintext attack. On the left there is an IA O_0 which when it is given m returns an encryption of m (using a fresh nonce). On the right there is an IA O_1 which when it is given m returns an encryption of a random message of the same length of m. The security notion asks that these two IAs look the same to any plausible adversary, i.e., $\mathsf{O}_0 \equiv \mathsf{O}_1$.

This definition is clearly not mathematically precise. It can be completely formalised, and if we do so, we get the notion of *indistinguishability under chosen-plaintext attack* (also known as *semantic security*), but the details of this are outside the scope of this course. The more formal definition is illustrated in Fig. **??** without going into details. It should be clear that if (G, E, D) is indistinguishability under chosen-plaintext attack, then the protocol in Fig. 5.7 securely implements SC. The simulator will simply encrypt random messages as in Fig. 5.6.

5.1.4 Exhaustive Search

Another consequence of reusing the same key many times is that the system can only be secure if the adversary's resources are limited.

More specifically, we have to assume that the adversary does not have enough computing power to run through all possibilities for the key k. This is because we have to take into account the possibility that the attacker finds out (by guessing or spying) one or more plaintext(s) m corresponding to ciphertext(s) c. Now he can execute the following simple algorithm known as *exhaustive key search*:

1. Initialize an empty list L.
2. For every possible key k': compute $D_{k'}(c)$ and check if $D_{k'}(c) = m$ for all the plaintext/ciphertext pairs (m, c) that are known to the adversary. If this is the case, add k' to the list L.

Of course, the correct key k will always be on the list L once the algorithm terminates. But if the list is very long, running the algorithm does not help the adversary much. However, a long list is unlikely if the adversary knows enough plaintext: say the key has t bits and the adversary knows u bits of plaintext in total. If we make the reasonable assumption that decrypting with an incorrect key gives you something random, then the probability that an incorrect key happens to produce a result that matches all u bits of plaintext is 2^{-u}. There are $2^t - 1$ incorrect keys, so we expect that $2^{-u}(2^t - 1)$ incorrect keys will survive the test. If $u > t$, then this is less than 1, so the adversary can expect that only the right key will survive the test. So we conclude:

FACT: If the adversary knows u bits of plaintext (and matching ciphertext), and u is larger than the length of the key, then we can expect that exhaustive search will identify the correct key.

It is important to understand that exhaustive search can be used against *any* system where the same key is used to send several messages, that is, any practical system. To make sure that exhaustive search is infeasible, a secure system these days must use keys of length about 128 bits or more – since doing 2^{128} repetitions of some non-trivial computation is currently considered completely infeasible[1]. Note that the key length by itself is no guarantee for security, it is only a necessary condition: given m, c, there might be a much faster way to find the key than simply trying all possibilities.

5.1.5 Price of an Attack

Thinking in terms of exhaustive search is part of estimating the cost of an attack and balancing that with the probability of an attack. Say that if someone breaks your system you will lose C dollars, and there is a probability p that an attacker can break into the system. Then the expected cost of attacks on the system is pC dollars. You want the amount pC to be negligible, i.e., something you don't mind paying. Let us for illustration say we find one dollar to be negligible. In that case

[1] At the beginning of 2018, the entire Bitcoin network was performing around 2^{64} "cryptographic operations" per second, meaning that it would take around 2^{64} seconds for the entire Bitcoin network to try all possibilities. The age of the universe is currently estimated to be $\approx 2^{58}$ seconds.

we would call a probability $p = 1/C$ that the system is broken into a negligible probability. In general, to know what probability you can consider negligible, you need to know the cost of a break each time the system is run and how many times it is run per day and for how long.

If we think like that, how long encryption keys should we use? Let us try to do a rather conservative estimate. The global computing power in the foreseeable future is about 2^{80} instructions per second. Imagine each instruction has some small probability p of causing a bad event that will cost you a million dollars, which is about 2^{20} dollars. Imagine this is repeated for 10 years, which is about 2^{28} seconds. Then the expected economic cost of the bad event over ten years would be $p2^{20+80+28} = p2^{128}$ USD. So if we say that an expected cost of one USD from a possible unlucky event over the next ten years is a negligible cost, then we could adopt $p = 2^{-128}$ as a negligible probability.

5.1.6 Stream Ciphers

Practical secret-key algorithms come in two flavors: *stream ciphers* and *block ciphers*.

A stream cipher is basically an algorithm X that expands a short key k and a nonce n to a much longer random *looking* string $X(k, n)$, which is then used to encrypt the message *as if* it was a one-time pad, that is, we take the output from the stream cipher and XOR it bitwise with the message m to get the ciphertext $c = m \oplus X(k, n)$. To decrypt, we compute $c \oplus X(k, n) = m \oplus X(k, n) \oplus X(k, n) = m$. Note that the receiver must know not only k but also n to do this. However, n is not secret so it can just be sent along with the ciphertext.

It is easy to see that the output string cannot really be random: say we expand a 128 bit key to a 1000 bit long output. We cannot possibly output more than 2^{128} different strings, whereas a really random 1000 bit string might take on any of the 2^{1000} possibilities. Even though the output string is not really random, the hope is that an adversary with limited computing power will not be able to tell the difference and hence cannot break the resulting encryption.

A stream cipher in practice will usually not generate the entire output at once, in fact one does not even need to know the required length in advance. Instead, one first initialises the algorithm with inputs k, n and one can then query the algorithm for the next bit or byte of the output string. This can in principle go on for as long as you wish.

This means that a stream cipher fits nicely with applications where the input to be encrypted arrives as a stream, say byte by byte and we do not know when it ends. An example is a user that types away at a keyboard. We can encrypt every byte immediately as it arrives, and send off the encrypted data.

A well known example of a stream cipher is RC4 which is often used in web browsers, which is rather unfortunate as it has some known weaknesses and should not be used. However, in recent years other, more and very fast secure stream ciphers have been proposed, some examples that can be found in the literature are SALSA20 and SNOW.

One should note that any good block cipher (which we look at in the following section) can also be used to build a secure stream cipher.

5.1.7 Block Ciphers

Block ciphers encrypt in their basic form a fixed size block of data, and output a block of the same size as the input. Examples are the former US standards DES (56 bit keys, 64 bit blocks), triple-DES (112 bit keys, 64 bit blocks) and the present standard AES (128 bit keys and blocks). Of these triple-DES is very widely used, for instance in banking applications, because the algorithm DES and triple DES are (or at least used to be) standard in all IBM security hardware. They are, however, partly outdated, in particular DES is completely insecure because the key is too short, and should not be used. AES is the new US standard that replaces DES and it uses (at least) 128 bit keys. Another example is the (patented) IDEA algorithm which is used in some versions of the PGP (Pretty Good Privacy, a partly free software package for general data encryption and authentication) products.

To use a block cipher in practice, one needs so called modes of operation, which are general methods that allow using a block cipher to encrypt a string of data of any length, and also to achieve security as required in Definition 5.3. Examples are Cipher Block Chaining (CBC) mode, Counter (CTR) Mode, and Output Feedback (OFB) mode. These modes need as input not only the key and the message, but also a nonce, as described above: at least one block of data that can change from one encryption operation to the next. In modes-of-operation lingo, the nonce is called an initialization vector, IV.

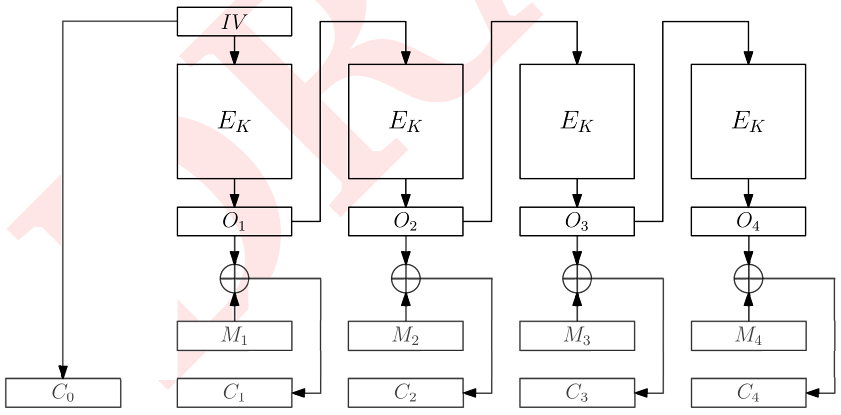


Figure 5.9 An illustration of OFB mode. In this example, four output blocks are produced. After this they are used for OTP encryption of the message. When used for encryption the IV is sent along to allow the receiver to regenerate the output sequence.

We mention for completeness that OFB mode is a mode that can be used to make a block cipher function in a way similar to a stream cipher. Taking AES with encryption function $\mathsf{AES}_K()$ as an example block cipher, we simply compute

$$\mathsf{AES}_K(IV), \mathsf{AES}_K(\mathsf{AES}_K(IV)), \ldots,$$

i.e. repeatedly feed the output block back into the encryption function (hence the name, Output Feedback). In this way we create a seemingly random stream of bits, as needed for a stream cipher.

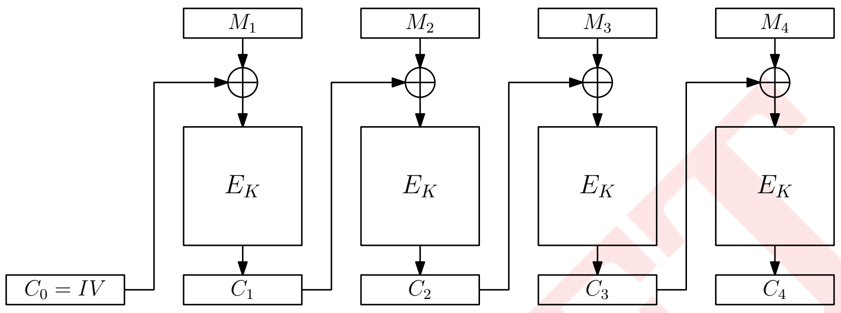


Figure 5.10 An illustration of CBC mode. In this example, four blocks are encrypted, producing a ciphertext with five blocks.

A more commonly used mode is CBC mode. Assume the message consists of 128 bit blocks $M_1, \ldots, M_t$, where we pad the last block i in some way, if it does not fill the required block length. Then the ciphertext will be $t+1$ blocks $C_0, \ldots, C_t$, where $C_0 = IV$ and for $i = 1, \ldots, t$,

$$C_i = \mathsf{AES}_K(M_i \oplus C_{i-1}) \ .$$

Here, $M_i \oplus C_{i-1}$ means bit-wise xor of the two blocks.

The final mode we describe is CTR mode, where again the message is $M_1, \ldots, M_t$, and the ciphertext depends on an IV. Again the ciphertext will be $t+1$ blocks $C_0, \ldots, C_t$, where $C_0 = IV$ and for $i = 1, \ldots, t$,

$$C_i = \mathsf{AES}_K(IV + i) \oplus M_i.$$

Here $IV + i$ means: think of IV as a 128 bit number and add i to this number modulo 2^{128} (or in other words, ignore any carry beyond bit position 128). CTR mode has the advantage over CBC that it is easy to encrypt several blocks in parallel, and that we only need the encryption function for the block cipher, even to decrypt.

As is apparent from the above modes, the IV is *not* secret, nevertheless it is important to get high quality encryption that it is chosen in the right way. The point is that the IV plays the role of the nonce, the extra input to the encryption we mentioned above, that is needed to ensure that the ciphertext will change,

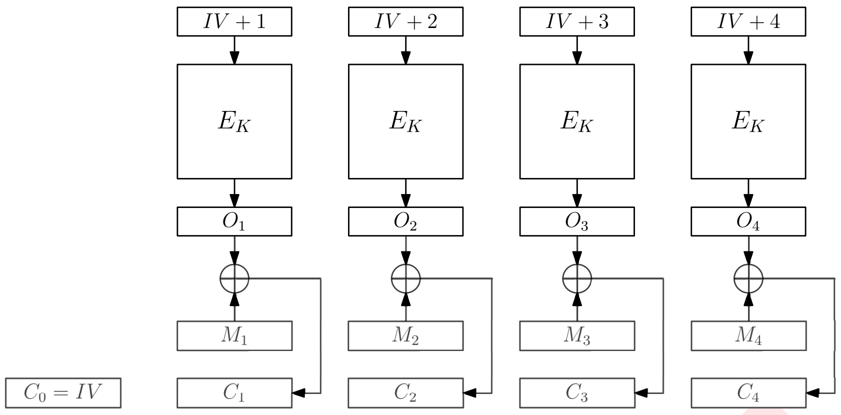


Figure 5.11 An illustration of CTR mode. In this example, four output blocks are produced. After this they are used for OTP encryption of the message. When used for encryption the IV is sent along to allow the receiver to regenerate the output sequence.

even if key and message are the same. Therefore the IV should be random, or chosen in some other way ensuring that values are reused with only very small probability. Unfortunately, several software implementations either ignore the problem of choosing IV, or circumvent it, for instance by fixing it to be the all-zero string. While this idea may make the implementation simpler, it degrades security and should be avoided.

In conclusion, as a rule of thumb, you will need a good block cipher, an appropriate mode of operation and a reasonable way to choose initialization vectors to get a useful encryption scheme.

Secret-key systems are generally quite fast, typically, one can get throughputs of 10-100 Mbytes/sec in software on a decent PC. And of course much more in a dedicated hardware implementation.

The basic problem with secret-key systems is that one must have the key in place at both the receiver and sender before sending data. This is not always an easy problem in practice (and is also the basic reason why no one wants to use the one-time pad in practice). The problem is particularly difficult if we have a large number of users where each user potentially may want to communicate privately with any other user. Then we need a separate key for each pair of users, and hence the number of keys increases quadratically with the number of users.

5.2 Confidentiality, Public-Key Systems

A public-key cryptosystem also has three algorithms G, E, D, for key generation, encryption and decryption. The algorithm G takes as input a key length κ and outputs a pair of keys of length κ.

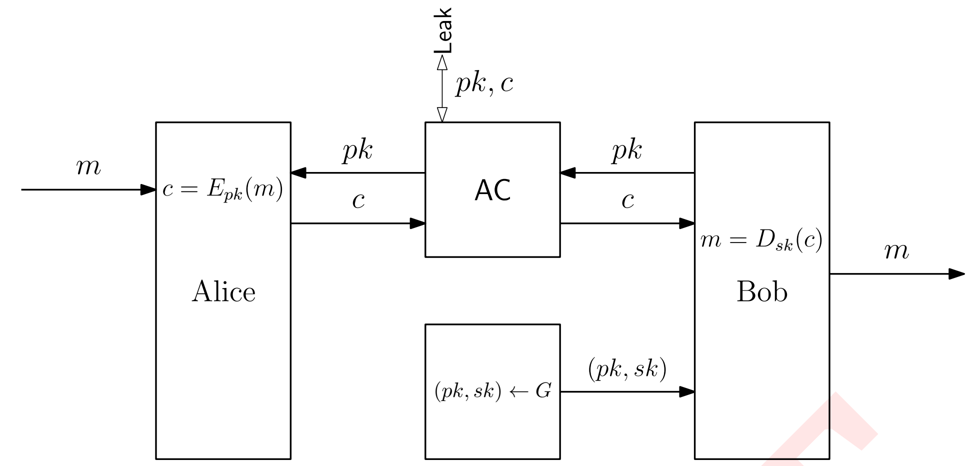


Figure 5.12 An illustration of how an asymmetric encryption scheme (G, E, D) is used. Alice has to send a message m to Bob over an authenticated channel which might leak the transmitted data. First Bob creates a key pair using G and sends pk to Alice. This public key is allowed to leak. When m arrives, Alice can compute and send $c = E_{pk}(m)$. This ciphertext is allowed to leak. When receiving c, Bob can compute $m = D_{sk}(c)$ (using the secret key) and output m. It is essential that sk is known only by Bob.

In contrast to secret-key systems, where the same key was used for both encryption and decryption, a public-key system makes use of a pair of matching keys, a public key pk and a secret key sk. The idea is that a user A of such a system will generate such a pair of keys in private on his own machine, will then publish pk, but keep sk private. For this to make sense, it clearly must be the case that even though there is a connection between pk and sk, it must be a very difficult computational problem to compute sk from pk.

The key generation algorithm is therefore usually more complex than for secret-key systems, because it has to produce pairs of keys with a carefully constructed relation between them. The goal is to ensure that anyone can encrypt a message m intended for A by running E on input m, pk to get $c = E_{pk}(m)$. Then A can decrypt this by running D on input sk, c because the system is constructed in such a way that for any message m we have:

$$m = D_{sk}(E_{pk}(m)).$$

We emphasize that, as for secret-key systems, the encryption algorithm may make random choices during the process, so that in fact the concrete value of $E_{pk}(m)$ may depend on those random choices, not just on pk and m. Indeed this is necessary for security as we will see in a moment.

Security here means that an adversary must have no idea about the plaintext corresponding to c, *even when given the public key pk*. The reason for this is that when using public-key cryptography everyone is allowed to know pk, and this of course includes the adversary.

We can formulate a notion of security for public key cryptosystems similar to what we did for secret key systems in Definition 5.3. The definition is exactly the same, with the only exception that the adversary is given the public key when the game starts.

Note that this immediately implies that the encryption function in a public-key cryptosystem must be *randomized* i.e., if one encrypts the same message twice then two different ciphertexts will come out. If this was not the case, it would be very easy for the adversary to win the game in Definition 5.3 in the following way:

1. The adversary submits a message m to the oracle;
2. The adversary receives a ciphertext c from the oracle;
3. The adversary encrypts $c' = E_{pk}(m)$ and concludes that c is an encryption of m if $c' = c$ and an encryption of a random r otherwise.

The definition is sketched in Fig. 5.13.

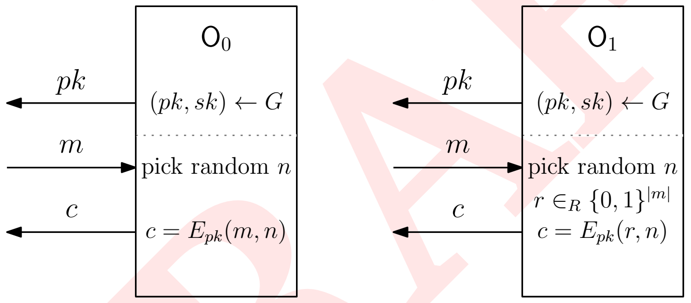


Figure 5.13 An illustration of the security notion indistinguishability under chosen-plaintext attack for public-key encryption. On the left there is an IA O_0 which hands out the public key and which when it is given m returns a random encryption of m. On the right there is a similar IA O_1 which when it is given m returns an encryption of a random message of the same length of m. The security notion asks that these two IAs look the same to any plausible adversary, i.e., $\mathsf{O}_0 \equiv \mathsf{O}_1$.

Public-key cryptosystems can also be broken using exhaustive search: Usually, there is only one private key corresponding to a given public key, so the adversary can just try all possible secret keys until he finds the one that matches pk. However, for all known public key systems, there are algorithms known for computing sk from pk that are *much faster* than just trying all possibilities. Therefore the size of keys for public-key systems are usually much bigger than they are for secret-key systems.

5.2.1 RSA

An important example is the RSA public-key system, where the public key consists of two numbers n and e, whereas the private key consists of numbers n and d. The number n is called the *modulus* and is the product of two prime numbers p, q. The numbers e and d are chosen to satisfy a particular relation that involves in an essential way the prime factors p, q. More precisely, one computes $d = f(e, p, q)$ for some particular easy to compute function f, the details of which we return to below.

It can be shown that computing the private key from the public one is as hard as solving the *factoring problem*, i.e., the problem of finding p, q from n. Conversely, it is clear that if one could factor n and find p, q, one could also compute d in the same way as when keys are generated. So breaking RSA is no harder than factoring.

We describe here the basic (or *vanilla*) version of RSA, which is deterministic and therefore insecure as described above. We will later discuss how to transform vanilla RSA into a *secure* public-key cryptosystem: in the basic version of RSA, messages are numbers in the interval $[0..n-1]$, and to encrypt a number m, one computes $c = m^e \bmod n$, i.e., we raise m to exponent e, divide by n and take the residue to be our result. In the same way, one decrypts by computing $c^d \bmod n$, and the special choice of e, d ensures that we always have

$$c^d \bmod n = (m^e \bmod n)^d \bmod n = m$$

As mentioned, the only known threat to this system is the possibility of factoring n. This is of course harder, the larger n is. The best known algorithms for factoring are capable of factoring a k-bit RSA modulus in time $2^{O(\sqrt[3]{k})}$ (very roughly and ignoring many details). The performance of these algorithms in practice is such that RSA keys must today have size at least 2000-3000 bits. But this size is not fixed for all times, it must increase with improvements in hardware and the increasing efficiency of the known factoring methods, and take into account how long into the future the encrypted data should stay private. For instance, when RSA was invented back in 1978, 500 bits long moduli were thought to be sufficient. In 2015, such a modulos could be factored in under 4 hours for 75$ using standard cloud computing platforms.

For completeness we now explain how to compute d from e, p and q, i.e., how to compute the function f from above: first, e must be chosen such that the greatest common divisor of e and $(p-1)(q-1)$ is 1. Then one computes d such that $ed \bmod (p-1)(q-1) = 1$. The condition on e ensures that a suitable d always exists. This is usually written as follows:

$$d = e^{-1} \bmod (p-1)(q-1).$$

At the time of writing, RSA is kone of the the most commonly used public-key systems in practice. One important reason for this is that when generating keys, it is possible to choose the public key (n, e) such that e is a small number, such as 3 (2 cannot be used since $(p-1)(q-1)$ is always even). Since the time needed to do the encryption or decryption is directly proportional to the number of bits in the involved exponent, this means that RSA encryption can be made several orders of magnitude faster. RSA is the only public-key system for which such an optimization is available. Note that a similar optimization cannot be used for the secret key (n, d), since d will be too easy to guess if it becomes too small.

5.2.2 Other Public-Key Systems

Other examples of public-key systems are the ElGamal crypto system based on the discrete logarithm problem. One way of implementing this system is based on a notion from algebraic geometry known as Elliptic Curves. ElGamal based on elliptic curves is known as *Elliptic Curve Cryptography* (ECC).

It is important to note that because of the different nature of the computational problems involved in different public-key systems, one cannot compare their security based on key sizes alone. In particular, systems based on ECC have the interesting property that no known attacks on them perform significantly better than brute force attacks. That is, a k-bit key can be broken in time $2^{O(k)}$, which is much slower than what can be done for RSA.

Therefore ECC can be used with much smaller keys than RSA. For this reason, in the last few years, ECC is quickly replacing RSA as the de-facto standard and is used in many places where small keys and ciphertexts are important.

Public-key cryptography based on both RSA and ECC is not secure if the adversary has access to a *quantum computer*. Many experts believe that the question is not *if* but *when* quantum computers will become a reality, and therefore cryptographers are currently working on designing *post-quantum* public-key cryptosystems, which are cryptosystems that are secure even against quantum computers. In particular, in 2017 NIST started an open competition to identify alternatives to RSA and ECC which can withstand attacks made by even quantum computers. Note however that the impact of quantum computers on the security of secret-key cryptosystems is very limited, and adjusting the key-length of current systems (e.g., switching from 128 to 256 bit keys in AES) is enough to ensure security even in a post-quantum world.

5.2.3 Using Public-Key Systems in Practice

Using public-key schemes in real applications brings up the following issues:

1. Most schemes, including the basic version of RSA described above, can be described as deterministic functions from plaintext and public key to ciphertext. As discussed above, this is not enough if one wants strong security guarantees, and therefore some randomness must be used during the encryption process, similar to the random choice of a nonce for secret-key encryption. As an example, consider the following modification of RSA: to encrypt a bit $b \in \{0, 1\}$, choose a random number r such that $r \bmod 2 = b$ (that is a random number whose least significant bit is equal to b). Now, encrypt r using the algorithm

described above. To decrypt, first recover r as described above and the compute $b = r \bmod 2$. Of course this method is not very efficient (since it encrypts a single bit at the time).

2. Public key systems are typically slower than secret key systems by several orders of magnitude, and hence cannot be used directly for bulk encryption of data. What one usually does is therefore to use the public key system just once in a session to communicate a key for a secret-key system, which is then used on the actual data. This is sometimes known as key enveloping. The problem now is that while a key for a secret-key system is usually quite short, like 128 bit for AES, a typical message in a public-key system is longer, like 2000 bits for RSA. So one needs a way to "pad" the AES key to obtain a plaintext block that can be processed by RSA.

   Simplistic ideas like just padding with 0's are not secure, since the theory for RSA gives the best security guarantees if the input numbers we encrypt are essentially random numbers in the interval $[0..n-1]$, which is of course not the case if the input is always only 128 bits long.

Further complications arise if the adversary can manipulate the input to the decryption algorithm and see how it reacts. Contrary to what you might think at first, this is a very realistic possibility – for instance, if the decryption algorithm is being used inside a web server. When setting up a secure connection with such a web server, it usually expects to receive a ciphertext, which it will decrypt and then communicate to the client whether the decryption was successful, and if not, what error occurred. This opens the possibility for an adversary to choose any string he likes and send this to the server, just to see whether the server can decrypt successfully, and if not, which error message it will send. This may reveal information on the secret key, and there are examples known where such an attack can actually break an encryption scheme. Such an attack is known as a *chosen ciphertext attack* (CCA). For instance, a previous version of the so called PKCS# 1 standard for use of RSA included a padding scheme, which was later broken under a CCA attack.

Fortunately, several international standards now have secure methods to do the padding, solving both of the above problems. One well known method is the Optimal Asymmetric Encryption Padding (OAEP) which is used in the default industry standard, namely the PKCS series of standards from RSA Laboratories we mentioned before. The details of this method are out of scope for this course, for us it is enough to know the following:

RSA Encryption with OAEP:

1. To encrypt a key K, one first computes a padded version $OAEP(K, R)$, where R is a string of random bits. The length of R is chosen as a function of the size of the RSA modulus such that $OAEP(K, R)$ is a number of suitable size for encryption under RSA.
2. The ciphertext is $OAEP(K, R)^e \bmod n$.
3. The receiver computes $(OAEP(K, R)^e \bmod n)^d \bmod n = OAEP(K, R)$, checks

that the format of the result is correct and if so, the receiver recovers K from $OAEP(K, R)$. The OAEP function is constructed such that this is easy.

The important fact for us is to know that usage of OAEP with RSA has been proven to be secure, even if the original plaintext size does not match the RSA block size, and even in a threat model where the adversary can do a CCA attack, i.e., play with the decryption equipment as sketched in the previous paragraph. The intuition here is that if the adversary tries to construct some ciphertext in some "illegal" way, decryption will always fail because the string coming out of the RSA decryption will not be in the correct format.

In summary: Public key schemes, including RSA, should never be used in practice without use of a secure padding method such as OAEP! Well designed software packages ensure this by default, but you cannot be completely sure unless you check.

Public-key systems solve the problem of distributing secret keys that we had before, but one still has to be careful: the sender of an encrypted message must make sure that the public key he is using actually belongs to the intended receiver. Otherwise someone else will be able to decrypt the message. Coming up with good solutions to this is less trivial in practice than one might expect – we return to this problem later, in the section on key management.

5.3 Exercises

Exercise 5.1 (One-time pad theory) A security officer in a bank has read a book on cryptology and learned that the one-time pad is completely unbreakable. So he insists that the payment orders sent from bank costumers over the net to the bank should be encrypted using the one-time pad (hardly a practical idea, but in this exercise we ignore the difficulties with distributing enough bits for the key). Assume that the format of such payment orders is as follows: the amount in Kroner to transfer from one account to another is stored in bits 0 to 20 of the string sent, least significant bit first.

Shortly after, an employee of the well known company Hackers Unlimited intercepts on the net an encrypted payment order. Based on the time at which it was sent, he guesses it contains a request to transfer his salary for next month to his account.

- Assume he makes less than a million kr per month. What is the value of bit 20 in the payment order? Show how he can modify the encrypted payment order in such a way that he will receive more than a million kr. extra next month.
- Is the security problem you have seen here a confidentiality problem or an authenticity problem?
- The notes claim that the one-time pad cannot be broken, and yet we have identified a security problem here. Why is this not a contradiction?
- A sender encrypts a message consisting of bits $m_1, \ldots, m_n$ with the one-time pad. Suppose an adversary intercepts the ciphertext and that he knows that m_i, the original bit at position i in the message, is 0 with probability p.

The adversary wants to modify the ciphertext such that the receiver will decrypt a 0-bit at position i in the message. Show that the adversary can make the receiver obtain a 0-bit in position i with probability $max(p, 1-p)$. Optional: show that the adversary cannot do better than $max(p, 1-p)$.

Exercise 5.2 Suppose someone misuses the one-time pad by using the same key K twice to encrypt two different messages M_1 and M_2 (of the same length). Show how an adversary can, from the two ciphertexts C_1, C_2, compute some information that depends *only* on M_1 and M_2, and not on the unknown key K. Do you think he might be able to even compute M_1 and M_2? Sketch some methods that might work.

Exercise 5.3 Consider the CBC mode of use for block ciphers. Describe how to decrypt a CBC ciphertext. Suppose a ciphertext consisting of many blocks is sent, but during transmission, 1 bit in a single block is changed. How large a part of the resulting decryption result might be incorrect? What if CTR mode is used instead?

Exercise 5.4 A manufacturer of windows (the kind you have in buildings, that is!) is about to launch a new product line: remote controlled windows that can be opened by just pushing a button on a little handheld remote control unit. The controller communicates with a window unit that can then open the window. Very useful for windows that are hard to reach physically. However, insurance companies have said that to insure objects in buildings with these windows, they will demand that it is ensured that a window can only be opened by the designated controller – and not by a controller that has been stolen elsewhere, for instance. Someone suggests that all communication between controller and window unit should be encrypted. Is this a good idea? Why or why not?

Exercise 5.5 Assume you have obtained some ciphertext that has been encrypted with the DES algorithm, and that you have also somehow obtained the corresponding plaintext. This means you can test whether a given DES key is the unknown key you are looking for. Assume you can test one key in time 10^{-10} seconds. How many days would it take to test all DES keys? What would the answer be if we had used AES instead?

Exercise 5.6 In view of the fact that the 56 bits of key in the DES algorithm is not quite enough it was suggested that one could just encrypt each plaintext block twice under two independently chosen keys K_1, K_2, that is, we would encrypt 64 bit block M to ciphertext C as $C = E_{K_2}(E_{K_1}(M))$, where E here stands for DES encryption. It seems that this is just a new block cipher, but now with $2 \cdot 56 = 112$ bits of key.

It turns out that this is not really true. Given enough memory space and a pair M, C of corresponding plaintext and ciphertext, one can break this cipher much faster than by trying all 2^{112} possibilities for the key. Here is the start of the algorithm:

1. Let variable K run through all possible 56-bit DES keys, for each possibility,

compute $D_K(C)$ (the decryption of C under key K). Store all these results in a sorted table.

2. ??

Fill in step 2, to obtain an algorithm that finds the correct key K_1, K_2 using a total of $2 \cdot 2^{56}$ DES encryptions and decryptions.

This so called *meet-in-the-middle attack* is the reason why DES these days is recommended for use only in the *two-key triple-DES* construction, where we set $C = E_{K_1}(E_{K_2}(E_{K_1}(M)))$. Would the attack above work against this cipher?

Exercise 5.7 Consider a cryptosystem where plaintext messages are sequences of letters from some alphabet – for concreteness let us assume the English one, which has 26 letters and that messages are written in English. A key is a permutation of the alphabet, i.e., a 1-to-1 mapping ϕ of the alphabet to itself. We encrypt by replacing the i'th letter m_i in the plaintext by $\phi(m_i)$. How many possible keys does this cryptosystem have? How does the number compare to the number of DES keys? Argue that it is not necessary to do exhaustive key search to find information on the plaintext from the ciphertext, by proposing one or more methods that might help to do it faster than by exhaustive search.

Exercise 5.8 This exercise is about why deterministic public-key encryption cannot be secure. Consider the following way to use RSA: the input is a sequence of letters in the English alphabet. Each letter is converted to a number such that a becomes 0, b becomes 1, etc. So we represent the plaintext by a sequence of numbers, each in the interval [0..25]. Finally, each number is encrypted individually using the RSA public key (n, e). Describe how this encryption scheme can be easily broken without having to factor n. Test your method by finding the cleartext for the ciphertext

$$365, 0, 4845, 14930, 2608, 2608, 0$$

and public key $(18721, 25)$, without factoring 18721.

Exercise 5.9 The public key for the RSA cryptosystem is a pair of numbers (n, e) where $n = pq$, the product of two primes p, q. The secret key is (n, d). Suppose we have $p = 3, q = 11$ so that $n = 33$ and messages are numbers in the interval [0..32]. It turns out that $e = 7, d = 3$ can be used. Encrypt the messages 2 and 5 and check that decryption works as expected.

Exercise 5.10 This exercise is about the problem that occurs when one needs to encrypt a secret 128-bit AES key under RSA. An RSA-block of data is typically at least 2000 bits long, because the bit-length of the modulus must be at least 2000 for security reasons. We need to convert the AES key to a longer block, and we shall see that if we do not take care we may get a security problem.

The most simple solution would seem to be to just pad with 0's, in other words if the key K consists of the bits

$$k_{127}, k_{126}, \ldots, k_1, k_0,$$

we convert this to

$$0, 0, \ldots, 0, k_{127}, k_{126}, \ldots, k_1, k_0,$$

adding enough 0's to reach the desired block length. Another way to say the same thing is that we just consider K as a binary number, which will certainly be in the required interval $[0..n-1]$ where n is the modulus, since n is at least 2000 bits long.

Show that if the public exponent is 3, then an adversary can easily compute K from the RSA ciphertext $K^3 \bmod n$. Hint: roots of integers can be computed very efficiently, even for large numbers.

Now suppose the public exponent is instead 17, and the modulus n has length 2150 bits. Show that the adversary can compute K from $K^{17} \bmod n$ while spending a factor roughly 2^{26} more time than in the previous question – which is certainly still feasible.

Exercise 5.11 (RSA and AES encryption) This exercise has two parts. The first asks you to implement RSA yourself. The second one asks you to try to use the Go implementation of AES.

1. Create a Go package with methods `KeyGen`, `Encrypt` and `Decrypt`, that implement RSA key generation, encryption and decryption. Your solution should use integers from the math/big package.

   The `KeyGen` method should take as input an integer k, such that the bit length of the generated modulus $n = pq$ is k. The primes p, q do not need to be primes with certainty, they only need to be "probable primes".

   The public exponent e should be 3 (the smallest possible value, which gives the fastest possible encryption). This means that the primes p, q that you output must be such that

   $$\gcd(3, p-1) = \gcd(3, q-1) = 1.$$

   Recall that $e = 3$ and d must satisfy that $3d \bmod (p-1)(q-1) = 1$. Another way to express this is to say that d must be the inverse of 3 modulo $(p-1)(q-1)$, this is written

   $$d = 3^{-1} \bmod (p-1)(q-1).$$

   This way to express the condition will be useful when computing d.

   Facts you may find useful:

   - Other than standard methods for addition and multiplication, `Mod` and `ModInverse` will be useful.
   - To generate cryptographically secure randomness, use `crypto/rand`. In particular, the function `Prime` from the `crypto/rand` package may be helpful to you here.

   Test your solution by verifying (at least) that your modulus has the required length and that encryption followed by decryption of a few random plaintexts outputs the original plaintexts. Note that plaintexts and ciphtertexts in RSA are basically numbers in a certain interval. So it is sufficient to test if encryption of a number followed by decryption returns the original number. You do not need to, for instance, convert character strings to numbers.

2. Implement methods `EncryptToFile` and `DecryptFromFile` that encrypt and decrypt using AES in counter mode, using a key that is supplied as input. The `EncryptToFile` method should take as input a file name and should write the ciphertext to the file. Conversely the `DecryptFromFile` method should read the ciphertext from the file specified, decrypt and output the plaintext. Test your solution by encrypting a secret RSA key to a file. Then decrypt from the file, and check that the result can be used for RSA decryption.

   Go?s official crypto package has an AES implementation, so importing "crypto/aes? does the trick. The key-size of the key provided to the `aes.NewCipher([]byte(key goes here))` call determines the strength of the cipher, so make sure its 16, 24, or 32 bytes, otherwise it will error (this is documented behaviour, but reiterating just in case).

Exercise 5.12 (implementing a secure channel I) Consider a network resource SC' defined as SC except that it does not leak $|m|$ on leak. It only leak a fixed message A MESSAGE WAS SENT. Prove that it is not the case that $\Pi_{G,E,D} \sqsubseteq \mathsf{SC}'$ when G, E, D is the one-time pad encryption scheme. Hint: consider the environment which flips a bit b and sends 1 of $b = 0$ and 11 if $b = 1$.

Exercise 5.13 (implementing a secure channel II) Consider the network resource SC in Fig. 5.5. Let $\Pi_{G,E,D}$ be the protocol from Fig. 5.12. Argue that when (G, E, D) is indistinguishable under chosen-plaintext attack then $\Pi_{G,E,D} \sqsubseteq \mathsf{SC}$.

6

Authenticity (DRAFT)

Contents

6.1 Authenticity, Secret-Key Systems

A secret-key system for authenticity consists of three algorithms G, MAC, V, where G generates a key, MAC is used to authenticate a message (MAC stands for Message Authentication Code), and V is used to verify a received message. G usually produces a key by just outputting a random bit string of fixed length. MAC takes as input a message m and a key K and outputs a message authentication code (MAC), $c = \mathsf{MAC}_K(m)$. Then the idea is to send m, c to the receiver who will run V on input k, m, c. The result $V_K(m, c)$ is either $\top$ (accept) or $\perp$ (reject).

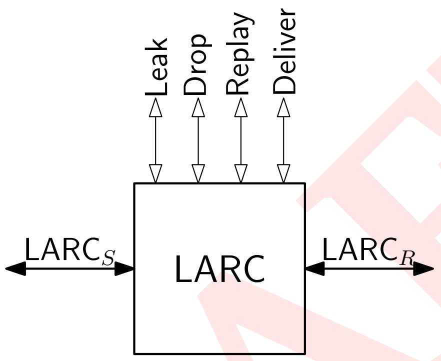
An authentication scheme must have the property that if no one tried to modify the message, the receiver will accept, i.e., we demand that

$$V_K(m, \mathsf{MAC}_K(m)) = \top$$

always. Of course, this condition says nothing about security. For this, we demand the following: the adversary is allowed to specify any number of messages he wants, say $m_1, ..., m_t$ and he is given valid MACs $c_1, ..., c_t$ for these messages. He is also allowed to specify pairs of form m, c and will be told if c is a valid MAC on m (he cannot decide himself as he does not know the secret key). We now have the following (informal) definition:

Definition 6.1 Consider any adversary who runs in time much less than what exhaustive key search would take. The authentication scheme is secure if no such adversary can play the game specified above and in the end produce a message m_0 and a MAC c_0 such that $V_K(m_0, c_0) = \top$ and $m_0 \notin \{m_1, ..., m_t\}$.

In more human language: even if the adversary has seen some number of messages with valid MACs, he still cannot come up with a message that was not sent and a valid MAC for that message. This is the case even if the adversary can choose the valid messages to be sent.



Ports The channel connects two parties with names S and R. For each party $P \in \{S, R\}$ it has a user port called LARC_P. It has a special port called Leak, which is used to model that the channel does not hide what is sent on it from its environment. It has a special port Drop, which is used to model that it can drop messages. It has a special port Replay, which is used to model that the adversary can inject a message which was sent earlier. However, the adversary cannot inject new messages. Finally, it has a special port Deliver which is used to model that it is its environment which controls when messages are delivered.

Init It keeps two sets InTransit and Send which are initially empty.

Send On input (R, m) on LARC_S, it outputs (S, R, m) on Leak and adds (S, R, m) to InTransit and Send.

Drop On input (S, R, m) on Drop it removes (S, R, m) from InTransit.

Replay On input (S, R, m) on Replay, where $(S, R, m) \in$ Send, it adds (S, R, m) to InTransit.

Deliver On input (S, R, m) on Deliver, where $(S, R, m) \in$ InTransit, it removes (S, R, m) from InTransit and outputs (S, m) on LARC_R.

Figure 6.1 Lossy Authenticated Reply Channel

Note that there is no requirement that the MAC algorithm should be randomized i.e., it is perfectly fine that, fixed a key K and a message m, a MAC scheme always outputs the same MAC c. (This was very different for encryption: given two ciphertexts it should be difficult to tell if they are an encryption of the same message or not).

6.1.1 Unconditional Authentication

Just as for encryption, there are unconditionally secure ways to do message authentication, that are, however, quite useless in practice. For instance, if there is a finite number of different messages that could be sent, then sender and receiver could agree in advance on a table containing for each message a random independently chosen t-bit MAC. The table then functions as the key. Of course, an adversary cannot find a correct MAC for a message that was not sent, except with probability 2^{-t}, regardless of how much computing power he has. Note that seeing other messages with legal MAC's does not help the adversary: Since the MAC's were independently chosen, faking the MAC of new message requires him to guess a random t-bit string.

This system has the property that the key gets larger, the more possible messages we have, and if we wanted to send, say m messages, we would need m tables, since just like for the one-time pad, we can not reuse the key. There are systems for unconditional authentication that are not quite as inefficient as this simple example, but there is no way around the fact that you need more key material, the more messages you want to be able to send.

6.1.2 Implementing an Authenticated Channel

We can use a secure MAC to implement an authenticated channel. In Fig. 6.1 we model an authenticated channel, with the slight problem that old messages can be replayed. This means that the channel does not allow to inject new messages, but if a message m was sent once, the adversary can inject it again later. We introduce this slight weakness to make it easier to implement LARC. If one later wants to implement LAC from LARC one could let the receiver keep track of which messages were already delivered. A disadvantage of this is that the receiver would then have to keep an ever-growing state, which we tend to avoid in distributed systems. We will discuss other options for mitigating replay attacks later.

Theorem 6.2 *Let* $\mathsf{MAC2LARC}_{G,\mathsf{MAC},V}$ *be the protocol in Fig. 6.2. If* (G, MAC, V) *is a secure MAC then* $\mathsf{MAC2LARC}_{G,\mathsf{MAC},V} \sqsubseteq \mathsf{LARC}$.

PROOF (INFORMAL) To prove $\mathsf{MAC2LARC}_{G,\mathsf{MAC},V} \sqsubseteq \mathsf{LARC}$ we have to come up with a simulator Sim such that $\mathsf{MAC2LARC}_{G,\mathsf{MAC},V} \equiv \mathsf{LARC} + \mathsf{Sim}$. A simulator Sim can simulate $\mathsf{MAC2LARC}_{G,\mathsf{MAC},V}$ given access to LARC simply by running the protocol "in the head" as in Fig. 4.6. When a message is dropped or delivered in $\mathsf{MAC2LARC}_{G,\mathsf{MAC},V}$ the simulator does the same on LARC. When a message is injected in $\mathsf{MAC2LARC}_{G,\mathsf{MAC},V}$ and Bob outputs m, then simulator will replay m on LARC. This will be allowed except when the adversary succeeds in making Bob

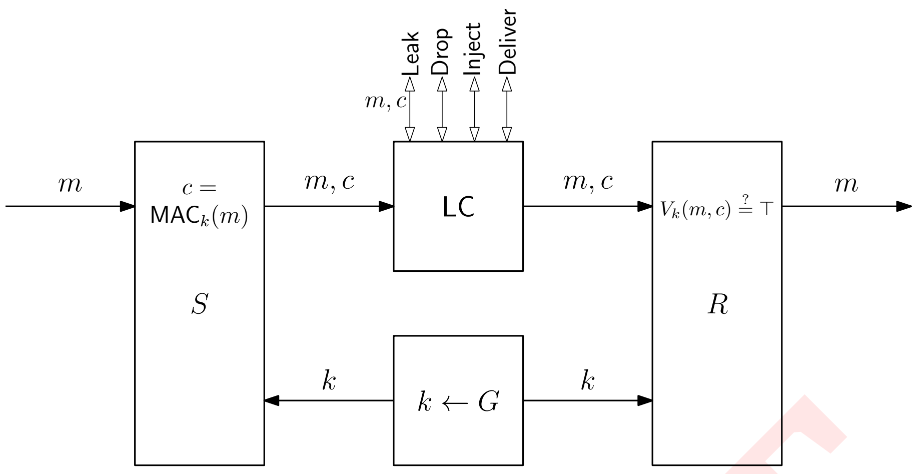


Figure 6.2 An illustration of how a symmetric authentication scheme (G, MAC, D) is used to send messages in an authenticated way. The protocol tries to implement a lossy authenticated replay channel LARC. Alice has to send a message m to Bob over an unauthenticated channel which leaks the transmitted data and which allows the adversary to inject messages. Somehow they managed to run the key generator G in the past and learn a common key K. When m arrives, Alice can compute and send a MAC $c = \mathsf{MAC}_K(m)$. This MAC is allowed to leak. When receiving m, c, Bob can check that $V_K(m, c) = \top$ and output m if this is the case. It is essential that k is known only by Alice and Bob. We call the protocol $\mathsf{MAC2LARC}_{G,\mathsf{MAC},D}$. To make Bob deliver a message m_0 never sent by Alice the adversary would have to inject a message (m_0, c_0) such that $\mathsf{MAC}_K(m_0, c_0) = \top$ without Alive having MACed m_0 and without knowing k. Security of a MAC scheme exactly states that this is hard.

deliver an m that was never sent by Alice. When the adversary is sufficiently computationally bounded and the MAC scheme is secure this happens with negligible probability. Therefore $\mathsf{MAC2LARC}_{G,\mathsf{MAC},V} \equiv \mathsf{LARC} + \mathsf{Sim}$. □

6.1.3 Practical systems and exhaustive search

For real-life applications, it is clear that a table as above will be much too large to handle. We need systems where we can use a small, fixed key, so that there will be much fewer possibilities for the key than there are possible messages. This means that after the adversary has seen a few valid messages and MAC's, the key is most likely uniquely determined: only a single key-value would lead to MAC's consistent with those the adversary has seen. The key can then be found by exhaustive search: the adversary simply runs through all possibilities for the key. For each candidate, he generates MAC's for all messages that were actually sent. If all these MAC's are identical to those that were sent by the legitimate sender, the adversary assumes he found the correct key. To ensure that such an exhaustive search is infeasible, the same constraints on key lengths (≥ 128) holds for this type of secret-key system as for the case of confidentiality. In addition, the MAC itself cannot be too short, otherwise an adversary might be able to simply guess a correct MAC for a falsified message. Usually, MAC's of 64 bits (and preferably more) are used.

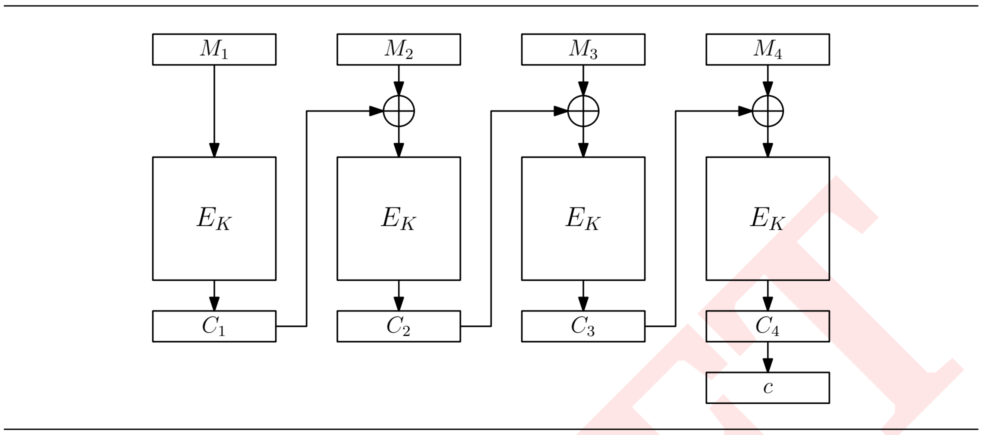


Figure 6.3 An illustration of CBC-MAC. In this example, four blocks are MACed. Only the last block is the MAC.

6.1.4 Example MAC algorithms

The two main examples of MAC algorithms are first a standard construction that allows to build a MAC algorithm from any secure block-cipher, and achieve the same throughput as the original system. Thus, if one belives in the security of for instance AES, a secure MAC algorithm follows immediately.

The construction is known as the CBC-MAC, where CBC as usual stands for cipher block chaining. The motivation for this name is straightforward: to compute such a MAC, one simply encrypts the message in CBC mode, as described in the previous chapter (using the all-zero IV) and defines the MAC to be *the last block of the ciphertext.* Such a MAC can be verified simply by recomputing the MAC from the received message and comparing to the received MAC. The intuition is that the last block of the CBC ciphertext depends in a complicated way on both the key and the entire message. Therefore any change to the message would result in a completely different last block that is hard for an adversary to predict. It can therefore be used as a MAC.

Another construction – quite popular in Internet standards – is known as HMAC, this one builds a secure MAC from any secure cryptographic hash function. Usually, the SHA1 hash function is used. We will talk about hash function in more detail later, here is it enough to know that the hash function is efficient to compute and produces a fixed size output (160 bits for SHA1), but complicated enough that it is hard to invert. In simplified form, HMAC on message m and key K is just $\mathsf{SHA1}(m||K)$, where $m||K$ means m concatenated by K. The actual construction applies (for technical reasons) some more complicated steps to obtain from m and K the string that is actually input to SHA1. In any case, the intuition is that since the hash function is hard to invert, the adversary cannot find K from the MAC and therefore does not know what to put into the hash function to compute a valid MAC for a modified message.

MAC algorithms are generally as fast as secret-key encryption, but suffer of course from the same key distribution problem that exist for any secret-key construction: we must have the secret key in place at sender and receiver before we can send anything.

6.2 Authenticity, Public-Key Systems

A public-key system for authenticity consists of three algorithms G, S, V, where S is used to authenticate(sign) a message, V is used to verify a received message, just like the secret-key case. However we here make use of the same key setup as in public-key encryption, i.e. we have a public key pk and a secret key sk. Sometimes we also call sk the signing key and call sometimes call pk the verification key (and denote it by vk). So, just for public-key encryption, G is more complicated than for MAC schemes because it must output a pair of keys with the right relation between them. The idea is that a user A of such a system will generate such a pair of keys in private on his own machine, will then publish pk, but keep sk private. Again, it must be a very difficult computational problem to compute sk from pk.

This setup can be used by A to send an authenticated message m by computing $c = S_{sk}(m)$ and send m, c. The receiver who (like anyone) knows A's public key pk can run V to get a result $V_{pk}(m, c)$ which is $\top$ (accept) or $\perp$ (reject) as before. Also as before, we must require that the receiver accepts if nothing happened to the message, i.e., for any message m and matching keys pk, sk we have

$$V_{pk}(m, S_{sk}(m)) = \top$$

On the other hand, it should be the case that an adversary who knows (of course) the public key of A and has seen a few message authenticated by A, cannot find any message m' that A never sent and a valid authenticator for it. The definition is similar to what we saw for the secret-key schemes:

Definition 6.3 The adversary is given the public key pk. He is allowed to specify any number of messages he wants, say $m_1, ..., m_t$ and he is given valid authenticators $c_1 = S_{sk}(m_1), ..., c_t = S_{sk}(m_t)$ for these messages. The public-key authentication scheme is secure if the adversary cannot efficiently produce a message m_0 and an authenticator c_0 such that $V_{pk}(m_0, c_0) = \top$, and $m_0 \notin \{m_0, ..., m_t\}$.

A system satisfying the above definition is said to be unforgeable under chosen message attack. It turns out that unforgeable under chosen message attack is

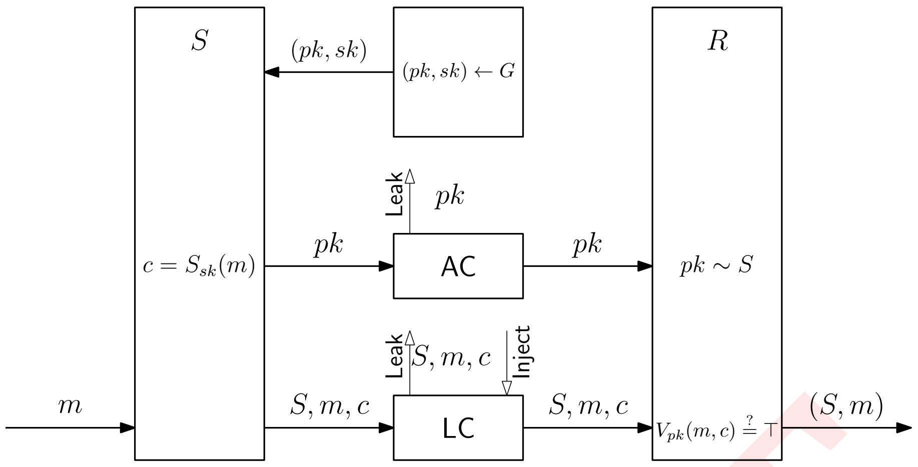


Figure 6.4 An illustration of how a digital signature scheme (G, S, D) is used to send messages in an authenticated way. The protocol tries to implement a lossy authenticated replay channel LARC. The sender, Alice, has to send a message m to a receiver, Bob, over an unauthenticated channel which leaks the transmitted data and which allows the adversary to inject messages. Somehow they managed to briefly have access to an authenticated channel at some point in time. Alice generates a key pair (pk, sk) and sends pk to Bob over the authenticated channel. Bob stores that pk is the public key of Alice. Later when a message arrives while Alice and Bob only has access to an unauthenticated channel LC, Alice can compute and send along a digital signature $c = S_{sk}(m)$. This digital signature is allowed to leak. Alice also sends along her identifier S. When receiving S, m, c, Bob can use S to fetch pk and check that $V_{pk}(m, c) = \top$ and output S, m if this is the case. It is essential that sk is known only by Alice. We call the protocol $\mathsf{Sig2LARC}_{G,S,D}$. To make Bob deliver a message m_0 never sent by Alice the adversary would have to inject a message (S, m_0, c_0) such that $V_{pk}(m_0, c_0) = \top$ without Alive having signed m_0 and without knowing sk. Security of a digital signature scheme exactly states that this is hard.

exactly what is needed for the protocol in Fig. 6.4 to securely implement LARC. We state this result without a proof.

Theorem 6.4 *Let* $\mathsf{Sig2LARC}_{G,S,V}$ *be the protocol in Fig. 6.4. If* (G, S, V) *is unforgeable under chosen message attack, then* $\mathsf{Sig2LARC}_{G,S,V} \sqsubseteq \mathsf{LARC}$.

6.2.1 Security of public-key systems

For similar reasons as with public-key crypto-systems, exhaustive key search is possible, but the adversary can usually find the secret key from the public one much faster than by exhaustive search. So keys must in general be much larger for public-key authentication than for the secret-key case.

6.2.2 Difference to secret-key: Transferability, Non-repudiation

Note here a very important difference between public and secret-key authentication: in the secret-key case, if B receives a message from A and accepts it, he is himself convinced that it came from A, but he will not be able to convince anyone else: both A and B have the secret key, and so anything B claims to have received from A could as well have been generated by B. This does not happen in the public-key case: the value $S_{sk}(m)$ can only be computed by someone who knows sk, and so it could only have been produced by A. This is why public-key authentication systems are also called digitial signature schemes, and the value $S_{sk}(m)$ is called A's signature on message m. The property is sometimes called non-repudiation

Note that, as for public-key encryption, it is important that the receiver uses the *right* public key when checking a signature, otherwise he can be made to believe that a message comes from A when this is not the case.

6.2.3 Examples of Digital Signature systems

It is usually the case that given some public-key *crypto-system*, the underlying techniques can also be used to build public-key signature schemes, though we usually need additional tools to get secure schemes.

Thus we have an RSA signature scheme, and also El-Gamal and Elliptic curve signature schemes. For instance, a first (but insecure) attempt to use RSA with public key $pk = (n, e)$ and private key $sk = (n, d)$ to generate signatures works as follows: we will assume that the message is a number m in $[0..n-1]$. The idea is that the signer who is the only one who knows the private key, will do something to the message that no one else could have done, namely apply the private-key operation to the message. That is, the signature on m is $S_{sk}(m) = m^d \bmod n$. It turns out that the special way e and d are chosen will ensure that raising $S_{sk}(m) = s$ to power e modulo n wil produce the original message, i.e., we have

$$s^e \bmod n = (m^d \bmod n)^e \bmod n = m$$

So when you receive a pair m, s you can check the signature by verifying whether $s^e \bmod n = m$, so for this system $V_{pk}(m, s)$ outputs accept if this condition is satisfied and reject otherwise.

In general, the signature schemes in the simplistic form mentioned here are the same speed as the corresponding crypto-systems. Thus they are much slower than the MAC algorithms, and too slow to be applied to really large amounts of data. One point to note, however, is that for RSA, if we apply the optimization where we use a small number as e, then checking a signature will be much faster than generating one. This is important in applications where a signature is generated once, but verified many times.

6.2.4 The problem with the simplistic scheme

Apart from the performance problem, the first attempt we just made to use RSA for signatures is actually insecure and does NOT satisfy the definition of security we gave earlier: an adversary could choose some s, compute $m = s^e \bmod n$ and claim that m is a signed message. Indeed, this will look OK, since m and s precisely relate as they should in order for a signature to be valid. This may not be a complete disaster, since the adversary cannot control the value of m – most likely m is not meaningful at all – but nevertheless this is an undesirable property, and it certainly means that basic RSA does not satisfy the definition of security we gave above.

6.2.5 Hash Functions

The solution to both the speed and security problems is known as cryptographic hash functions. Such a function h should have the following properties:

- It should be able to take a message of any length as input.
- It should produce an output of fixed length
- It should be fast to compute, speed similar to the best secret-key systems
- It should be a hard computational problem to produce a collision: two inputs x, y such that $x \neq y$ but $h(x) = h(y)$. Note that the first two conditions clearly imply that many such x, y *exist*. But this is fine, we just need that it is hard to find such pairs.

W.r.t. the last condition, one can of course always just try to compute h on random x's in the hope that this will happen to produce a collision. If the output of h has length k bits, this can be expected to succeed after about $2^{k/2}$ evaluations of h.

We argue informally why this is the case: the reason is the so called "birthday paradox" (which is not actually a paradox, but just a somewhat surprising fact from probability theory). If we choose a random (long enough) input string x and compute $h(x)$, it is reasonable to assume that this will be a randomly chosen k-bit string. Do this for x_1 and x_2, then of course the probability that $h(x_1) = h(x_2)$ is $1/2^k$. But what if we choose t values, what is then the probability of a collision, i.e., that at least two of the values $h(x_1), ..., h(x_t)$ are equal? the observation is that there are in fact $t(t-1)/2$ *pairs* of values, and for each such pair, there is a chance of 2^{-k} that the two values involved in this pair happens to be equal. This means that $\frac{t(t-1)}{2 \cdot 2^k}$ is an estimate for the probability of a collision. This is not quite accurate, but is OK for small t. As t increases, the number of pairs is about t^2, so this is why the probability of a collision becomes substantial when $t^2 \approx 2^k$, i.e., when $t \approx 2^{k/2}$.

Conclusion: A hash-function wiht k-bit output has, at best, $k/2$-bit security, in the sense that a brute force attack will find a collision in time about $2^{k/2}$ evaluations of the function.

In order to make such "birthday attack" infeasible, good hash functions must have output length at least 160 bits, since it is believed that currently evaluating a non-trivial function 2^{80} times is infeasible. Even this is changing, however, and there is currently a tendency towards choosing larger values, typically 256 bit outputs.

Well-known examples of hash functions are the European design RIPEMD-160 and the US standard SHA-1 and MD5. Of these, RIPEMD never had a commercial breakthrough, and MD5 has recently been broken completely. Consequently SHA-1 has been used in many practical applications until recently. But even SHA-1 has security problems: some years ago, a Chinese research group has found a method that will find collisions for SHA-1 in time about 2^{69} evaluations of the function. This is much less than the brute force attack which would be 2^{80}, on the other hand the attack did not seem realistic at first. However, in spring 2017 a Dutch team used a highly optimised version of the attack to actually find collisions for SHA-1.

Therefore, most applications are switching or have switched to other standardized functions with longer outputs, such as the “big brother” of SHA-1, namely SHA-256, which produces 256-bit outputs. Finally, the US standards organization NIST published in 2015 a new standard hash function called SHA-3 which can produce outputs of up to 1600 bits. This function was selected after a “competition” among several candidates proposed by international researchers. SHA-256 and related functions are still standardized, but since both the MD family and the previous SHA constructions are based on the same design principle it was felt that was a need for a completely different approach.

6.2.6 Hash-and-Sign Signatures

The basic idea for using a hash function h in connection with signatures is that computing $h(m)$ produces a “fingerprint”, or a “digest” of m, where no one can find any other message with the same fingerprint (this would lead to a collision). Therefore, intuitively, signing $h(m)$ is just as convincing as signing m itself. In practice what we do is to fix a hash function h to be used by all users, and then if we have a signature scheme such as basic RSA with signature algorithm S, we define a new signature scheme with signature algorithm S', where we define that the signature on message m to be $S'_{sk}(m) = S_{sk}(h(m))$. In other words, we first hash the message and then we sign the hash value[1]. Accordingly, we also define a new verification algorithm V', namely on input message m and signature s, $V'_{pk}(m, s)$ will compute $h(m)$, execute $V_{pk}(h(m), s)$ and accept if and only if V said accept.

Because $h(m)$ has fixed length and is usually much shorter than the message, this solves the speed problems we had before, and even in some cases can help to make a scheme more secure than if we signed messages directly.

[1] In some cases, such as with RSA, the hash value is much shorter than the size of blocks the signature scheme would naturally work with. In such cases some padding is needed to make the sizes fit.

For instance, the attack above on simplistic RSA where the adversary chooses a random "signature", applies the public key and claims the result is a signed message, does not work if messages are hashed before they are signed. The reason is as follows: of course, the adversary can still choose some s and compute $s^e \bmod n$. With basic RSA, this could be claimed to be a message. But with the "hash +RSA" scheme, the adversary instead has to find m such that $h(m) = s^e \bmod n$. This is the problem of inverting h, i.e., given some value (such as $s^e \bmod n$) in the image of the function, find a preimage of this value. It turns out that this is a hard problem: one can show that if it is hard to find collisions for h it is also hard to invert it (a function which is hard to invert is typically called a one-way function). See the exercises for a proof of this.

In the rest of the book, when we talk about "RSA signatures", we will always mean a secure variant, where hashing is used, even if we do not refer explicitly to the hash function used.

6.2.7 Using Digital Signatures in Practice

We end this section with some thoughts on usage of digital signatures in real life. In particular, can digitial sigatures have legal value, just like hand written signatures? The answer is yes, in fact the legislation in many countries is prepared for this, but some conditions need to be met. Some are obvious, such as using large enough key sizes, and storing the private key securely. Other things are more subtle: clearly, it should be possible for an honest user to report it if his private key has been stolen or compromised, and we then have to assert that all signatures that match his old public key should now be considered invalid. Note here that even documents with an older date cannot be trusted anymore since using the private key, one can forge documents with any desired date.

However, since digital sigtures can be used to bind the signer to some obligation, it may be in a cheating signer's interest to report his private key stolen even if it is not, just to repudiate a signed document. There are several solutions to this. One solution is to have an independent time stamping service, who is asked to countersign important documents. Since the signature of such a service remains valid, even if users have their keys compromised, this would solve the above problem that even old documents cannot be trusted when the private key is stolen.

A supplementary solution which is virtually always necessary, is to have paper based documents with hand-written signatures where users agree to certain regulations for the use of digital signtures. For instance, a user typically has to agree the he assumes responsibility for his own private key, including reporting it stolen as soon as he suspects its security is harmed. This means one can have rules stating that if a user does not protest against the consequences of having digitally signed something before some deadline, he cannot get out the obligation by claiming that the private key was compromised. Such agreements resemble a lot the rules that apply to credit cards, and are used in all homebanking systems.

A final thing to note about practical applications of signatures is that

A digital signature commits the signer to exactly what was signed and nothing else.

While this seems blatantly obvious when stated explicitly as here, it is surprisingly often overlooked in applications: suppose, for instance, we have a system for signing e-mails. Is there a difference between the case where only the message content is signed and the case where all fields of the mail, including the intended recipient's address, are signed? despite what you may think, there is in fact a very big difference. The latter method ensures that the sender always commits to the identity of the recipient, the former only does this if the recipient is explicitly mentioned in the text of the mail. As a silly –but illustrative– example: suppose Bob is in love with Alice and sends her a signed mail saying "I love you, will you marry me?". Then with the former method, anyone can claim to have a signed offer of marriage from Bob!

A more subtle problem of this type is the case when a sender encrypts a message and then signs the ciphertext. This signature does not commit the sender to the plaintext message! The simple reason is that anyone can eavesdrop a ciphertext from the net and sign it, without knowing anything about the plaintext.

6.3 Replay attacks

The secret-key and public-key approach to authenticating messages are actually not satisfactory by themselves in all cases. The reason for this is that if I receive for instance a message m with a digital signature from A, this only proves that *at some point*, A produced this message. It leaves open the possibility for an adversary to take a copy of the signed message and send it to me as many times as he wants. This is known as a replay attack. If A is a bank customer, the reciever is a bank, and the message is a request to transfer money from A's account to someone else, the security problems with replay should be evident.

So what we really want in practice is often not just to protect the integrity of messages, but to have a real authentic channel, that is, we want B to receive the exact same sequence of messages that A sends, and if this not possible, we want to come as close to this as we can. If we do not have physical control over the communication line, we cannot prevent an adversary from physically blocking some messages or from reordering them before they are received, but we should at least be able to make sure that replayed messages are filtered out.

One trivial way to ensure this is have the sender make sure that he never sends exactly the same message twice, for instance by appending a sequence number, and also add a MAC (computed over both message and sequence number. Then we can have the receiver store every message he ever receives (or at least the sequence numbers). This will allow the receiver to filter out every replayed message, and also to correctly place all messages he gets in the order they were sent.

Of course, this is hardly a practical solution. Even if we do use sequence numbers, we cannot expect the receiver to store more than the sequence number of a small number of recently accepted message. Suppose, for instance, that he stores only the last accepted message. Then if the receiver has stored number n and the next message does not have number $n+1$, we need a rule for the receiver to decide what to do. The problem is that in some cases, messages may get lost or be reordered even if no attacker is present. One possibility is to require that the next message must have number $\geq n$ to be accepted. This will prevent replay, but may also mean that good messages get rejected if they arrive too late.

A different approach appends a timestamp to messages. To be of any use, this of course requires that the receiver checks the time stamp. Again, one may in principle store all received time stamps. This leads to perfect security against replay, but is usually not practical. Another method is to check time stamps against your own system time, to ensure that it is not too old. Some compromise has to be made here, so that on one hand good messages are accepted, but on the other hand replayed messages are rejected. This requires that there is some synchronization between sender and receiver, and also that messages are delivered quickly enough. On the other hand, there is no need here to remember any information about earlier messages.

Finally, one may use interaction: we can have the receiver first send a number R to the sender. This number can be chosen at random, or be a sequence number, the only real requirement is that it has never been used before (i.e., it is what we called a nonce earlier). Then the sender sends the message plus a MAC computed over the message and R. This will prevent replay, and there is no need for sychronization, but one does need to at least remember some state in order to ensure that R-values are not reused. In addition this is not trivial to implement securely if the receiver has to handle several connections in parallel – for instance, he has to keep track of which R-values belong to which connections.

6.4 Getting both Confidentiality and Authenticity

Many applications require both confidentiality and authenticity at the same time. One may of course use a combination of the two types of techniques we have seen. But it is worth while to warn against some pitfalls:

In the secret-key case, we will want to compute MAC's and also encrypt messages. But one must use different and independent keys for the two purposes: there is NO guarantee for security if the same key is used.

However, there exists specially engineered encryption modes that provide both confidentiality and authenticity at the same time, using one key. Such encryption schemes are known as authenticated encryption, and are becoming increasingly popular in practice, since it is simpler to use a single scheme that provides both confidentiality and authenticity than having to use two different tools at the same time. Examples of authenticated encryption schemes are OCB, GCM and CCM.

Another point is that if you first authenticate by computing a MAC or a signature on the plaintext, and you then encrypt, you have to encrypt the MAC or signature as well. This is because the MAC or signature may in general contain information about the plaintext. If this was left in the clear, we would not have hidden the message completely.

This method of appending a signature and then encrypting the whole thing has the potential problem that we are giving the adversary for free some information on the structure of the plaintext that is encrypted, i.e., it is always of form message followed by a signature or a MAC. This can be a problem if the encryption is not completely secure. Therefore, it is often recommended to first encrypt and then compute a MAC on the ciphertext.

6.5 Exercise

Exercise 6.1 (Enc-then-sign or Sign-then-encrypt) You are consultant to a company that offers access to a database D. Different users have access to different parts of the database and moreover, the database contains sensitive information. Therefore the security policy says:

- When a request for information arrives, D should be able to determine which user sent the request.
- A user should not be able to get get information about which data other users asked for.

We can assume that every user A has a private RSA key sk_A of his own, and the database stores a list of public keys of all users. Also all users know the public key pk_D of D.

The threat model assumes that the database is sufficiently protected that it will not be hacked, but network traffic can attacked and manipulated.

One suggestion to ensure that the policy is followed is: when user A wants to send a request R to D, he will send

$$E_{pk_D}(R), S_{sk_A}(E_{pk_D}(R)), A$$

to D, that is, encrypt the request under D's public key, append his signature on the encrypted request, and then append his username. D will look up pk_A, verify the signature, and if it is correct, will decrypt the request, and return an answer of form $E_{pk_A}(data)$, that is, the answer to the request, encrypted under A's public key.

Another suggestion goes as follows: when user A wants to send a request R to D, he will send

$$E_{pk_D}(R, S_{sk_A}(R)), A$$

to D, that is, append his signature on the request, encrypt this under D's public key, and then append his username. D will decrypt the request and signature, look up pk_A, verify the signature, and if it is correct, will return an answer of form $E_{pk_A}(data)$, that is, the answer to the request, encrypted under A's public key.

One of these solutions fails to satisfy the security policy. Which one, and why? are there any general conclusions one could draw from this example?

Exercise 6.2 Many countries have restrictions on export of soft/hardware for cryptography. But often, systems that provide authenticity only are not under these restrictions. It seems that software providing encryption is thought to be more "dangerous", should it fall into the hands of criminals. Rivest (the R in RSA) has made an observation showing that such a distinction between systems for authenticity and for confidentiality makes very little sense: the distinction relies, of course, on the assumption that systems for authenticity cannot be "misused" to provide confidentiality. This is of course always possible if one can reverse engineer the soft- or hardware, but one could argue that this is too cumbersome/expensive for an attacker. What Rivest observed is that such "misuse" is always possible *without any reverse engineering and without using additional cryptography*:

Suppose you are given a system providing secret-key authentication. In other words, the sender and receiver have access to a (soft or hardware) module containing a shared key K. The module will accept any message M as input and will return a MAC $\mathsf{MAC}_K(M)$. It will also accept as input a message and a MAC and will tell you if the MAC is valid w.r.t. the message and the current key. You cannot modify the modules to do anything else, nor can you modify the key. You don't have any other sources of cryptography. Show how to build from this a system that provides confidentiality. You may assume that given a message M and a value C, it is infeasible to decide if C is a valid MAC or a random value (unless of course you know the key).

Exercise 6.3 For several MAC constructions, one needs to split the input message into blocks of some fixed size, in this exercise we assume 128 bits/block for concreteness. A padding method is then needed to append some bits to the message in order make its length after padding divisible by 128. Consider the following method: "If the message length is divisible by 128, do nothing, otherwise append a minimal number of zero bits so that the resulting length is divisible by 128".

Explain why this method would be insecure, no matter which MAC algorithm we use. I.e., show how a MAC on one message can be turned into a MAC on a different message without using the secret key.

Consider instead the following: "If the message length is divisible by 128, do nothing, otherwise append a minimal number of zero bits so that the resulting length is divisible by 128. Finally, append a block containing in binary the bit-length of the message".

Explain why this method does not suffer from the problem(s) you identified with the first method. Note: you are not expected to prove that, for instance, CBC-MAC is secure with this second padding scheme, this is not even known to be true in all attack models. You are only expected to argue why the generic problem with the first scheme is solved in the second.

Clearly, the second method makes the message one block longer than we would need if we just wanted to store the message in an integral number of blocks. Is it possible to have a padding method that never makes the message longer in this way, and yet does not have the problem of the "only-zero" padding?

Exercise 6.4 Both the SHA-1 and the SHA-256 hash functions are constructed using the same basic design idea: first, a so called compression function f is constructed, that takes as input two strings m and s, of lengths 512 bits and 160 bits, respectively. Next, a 160 bit block s_0 is fixed as part of the description of the hash function. Now, given a message M to hash, we split it in 512 bit blocks $m_1, ..., m_t$, padding the last block with 0's if it is incomplete. Finally a block m_{t+1} is appended that contains the original length of the message, written in binary. We now compute, for $i = 1, .., t+1$, $s_i = f(m_i, s_{i-1})$. The final hash value is defined to be s_{t+1}.

Show that if one can find a collision, i.e., different messages M, M' producing the same hash value, then one can easily find a collision for the function f.

Thus if we believe it is hard to find collisions for f, we have to believe it is hard as well for the entire hash function.

Hint: When we hash M, we process blocks $m_1, ..., m_{t+1}$ to get values $s_1, ..., s_{t+1}$. When we hash M', we process blocks $m'_1, ..., m'_{t'+1}$ to get values $s'_1, ..., s'_{t'+1}$. Note that if the hash of M is the same as the hash of M', this means that we have

$$f(m_{t+1}, s_t) = s_{t+1} = s'_{t'+1} = f(m'_{t'+1}, s'_{t'})$$

This may immediately give a collision for f, and we are done. If not, then $(m_{t+1}, s_t) = (m'_{t'+1}, s'_{t'})$. What can you conclude from this?

Exercise 6.5 This exercise is concerned with proving the fact that if a hash function h is collision intractable then it is also hard to invert h (a function which is hard to invert is typically called a one-way function). Here, by inverting h, we mean the problem where we are given a in the image of h and we need to find x such that $h(x) = a$. In this exercise we will assume for simplicity that $h : \{0,1\}^{k+1} \mapsto \{0,1\}^k$, and that $Im(h) = \{0,1\}^k$. In other words, h maps $k+1$-bit strings to k bit strings and h is surjective. Both restrictions can be removed quite easily.

To show that inverting h is at least as hard as finding collisions, we assume that we are given an algorithm Alg that inverts h, i.e, on input $a \in Im(h)$ it returns x such that $h(x) = a$. We then construct an algorithm that will use Alg to find collisions with good probability, as follows:

1. Choose $y \in \{0,1\}^{k+1}$ uniformly at random and compute $a = h(y)$
2. Execute Alg on input a and let x be the output
3. Output x, y.

Does this work? Note first that it is always the case that $h(x) = a = h(y)$, but for (x, y) to be a collision, it must also be the case that $x \neq y$. To estimate the chance of this, do the following:

1. Let $h^{-1}(a)$ be the preimage of a, i.e., the set of inputs that are mapped to a by h, and let $|h^{-1}(a)|$ be the size of this set. Show that when we choose y uniformly as above and set $a = h(y)$, then $Pr(|h^{-1}(a)| \geq 2) \geq 1/2$.
2. Let E be the event that $|h^{-1}(a)| \geq 2$. Show that $Pr(x \neq y | \ E) \geq 1/2$.
3. Show that $Pr(x \neq y) \geq 1/4$.

Hence the probability that our algorithm successfully produces a collision is at least 1/4. Hint for item 1): construct a set $A \subset \{0,1\}^{k+1}$ as follows. For every element a in the output set $\{0,1\}^k$, choose an input u such that $h(u) = a$ and add u to A. Note that now A has exactly 2^k elements. Therefore, when you choose y uniformly it will be the case that $Pr(y \notin A) = 1/2$.

Exercise 6.6 Alice has a PC and is worried that someone might get access to her machine and modify some important files she has while she is away on holiday. In order to at least be able to detect any such changes, she uses a hash function h, for which it is infeasible to find collisions. For each relevant file, she applies h to the contents of the file and stores all output values in a separate directory on her hard disk. When she comes back, she will verify that the hash values on the files still match the values she stored. Explain why this solution is unlikely to work.

Alice thinks again and comes up with two solutions that seem better:

1. She stores the hash values on a USB stick that she takes with her, when she comes back she verifies the hash values on the stick as above.
2. She chooses a secret key K, and for every file f, she computes $\mathsf{MAC}_K(f)$ using some secure authentication scheme. She stores all the MAC values in a special directory and brings K with her on a USB stick. When she comes back she verifies all the MACs.

Will both solutions work if Alice goes on holiday once? Is the answer the same if she does it twice or more?

Exercise 6.7 Many car companies offer remote controls for their cars, so that the car can be remotely unlocked. One potential security problem is that since the communication between car and remote control is wireless, it is quite easy using standard equipment to eavesdrop the communication relating to a target car. An adversary could try to use the information obtained this way to later trick the car into unlocking. Design one or more methods for handling the communication between car and remote control that would be secure against this threat model.

Exercise 6.8 Here is a suggestion for doing message authentication based on public key technology, but without hash functions: suppose the receiver B knows the sender A's public key pk_A. B chooses an AES key K, and sends to A the encryption $E_{pk_A}(K)$ (concretely, this could be RSA plus OAEP). A will decrypt to get K, and then sends to B the message M plus the message authentication code $\mathsf{CBC} - \mathsf{MAC}_K(M)$.

Explain why this cannot be used as a digital signature scheme, even if we assume that both the public-key encryption and the CBC-MAC construction are secure.

Exercise 6.9 Another reason why hash functions are necessary ingredients in signature schemes: Suppose we use RSA for digital signatures, but without using a hash function. Concretely, this means that every message m is a number in the interval $[0..n-1]$ where (n, e) is the public key and (n, d) is the private one. The signature on m is $m^d \bmod n$. Suppose an adversary is given valid signatures s_1, s_2 on messages m_1, m_2. Show that the adversary can now easily generate on his own a valid signature on $m_1 \cdot m_2 \bmod n$, namely $s_1 \cdot s_2 \bmod n$ is a valid signature on $m_1 \cdot m_2 \bmod n$. Hint: for any numbers a, b it holds that

$$((a \bmod n) \cdot (b \bmod n)) \bmod n = ab \bmod n.$$

Exercise 6.10 (RSA signatures) Extend your Go package from Exercise 5.11 so that it can generate and verify RSA signatures, where the message is first hashed with SHA-256 and then the hash value is signed using RSA, as described in Sec. 6.4. The hashing can be done with the `crypto/sha256` package.

Note: international standards for signatures always demand that the hash value is padded in some way before being passed to RSA, but you can ignore this here. Thus the hash value (which will be returned as a byte array) can be converted to an integer directly. Such direct conversion should not be done in a real application.

In addition to the code, your solution should contain the following:

1. Verify that you can both generate and verify the signature on at least one message. Also modify the message and check that your verification rejects.
2. Measure the speed at which you can hash, in bits per second. For this you should time the hashing of messages much longer than a hash value, in order to get realistic data – say 10KB;
3. Measure the time you code spends to produce an RSA signature on a hash value when using a 2000 bit RSA key;
4. Assume you had to process the entire message using RSA. Use the result from question 3 to compute the speed at which you could do this (in bits per second). Hint: one of the RSA operations you timed in question 3 would allow you to process about 2000 bits. Compare your result to the speed you measured in question 2. Does it look like hashing makes signing more efficient?

Exercise 6.11 Suppose you are given a hash function h that accepts input strings of length 256 bits and produces 128 bit outputs. We assume h is a good cryptographic hash function: h can be computed efficiently, but it has the basic property required for hash functions, namely it is not in practice possible to find inputs $x \neq y$ such that $h(x) = h(y)$, a so-called collision for h. Consider the hash function H defined as follows: it takes as input a string X of length 512 bits. We split X in two strings X_0, X_1 each of length 256 bits, such that X is the concatenation of X_0 and X_1, written $X = X_0||X_1$. We define

$$H(X) = h(h(X_0)||h(X_1))$$

Show that if one could find a collision for H, i.e., $X \neq Y$ such that $H(X) = H(Y)$, one could easily find a collision or h as well. Therefore, finding a collision for H is as hard as finding one for h. Generalize this idea to construct from h a secure hash function that takes inputs of length $2^j \cdot 256$ bits, for any j, and argue that it is hard to find collisions for your function, if it is hard find them for h.

Exercise 6.12 Suppose A and B are connected by a channel that will always eventually deliver any message sent, but may delay any message arbitrarily. There is also an adversary that may attempt a replay attack, so he is allowed to inject a copy of a message that was received before. Assume the following:

- We use a secure signature scheme system for authenticity. A signs every message he sends to B.
- A never sends exactly the same message twice, we assume sequence numbers are used to ensure this: A keeps a counter, for every message to be sent, the counter value is appended to the message and the counter is incremented.
- B can store information about at most N sequence numbers he received (and accepted) from A earlier. Thus, if all N slots are full and he receives an additional message, he cannot store its sequence number, unless he deletes one of those he already has in memory.

For every message B receives, he must decide –at the time of reception– based on the information he stores whether to accept or reject the message. A message delivered to B is said to be genuine if it was sent by A and has not been accepted by B before. It is a replay if it has been accepted by B before. A strategy for B is said to be *secure* if it ensures both that all genuine messages are accepted and that all replayed messages are rejected.

- Give a secure strategy for B, assuming that A never sends more than $N + 1$ messages.
- On the other hand, show that if A sends $N + 2$ or more messages, there is no secure strategy B can apply.
- Assume now that the channel does not reorder messages arbitrarily. Instead, we assume the following: when message number i is delivered and sequence numbers $j_1 < j_2 < ... < j_N$ are the N largest sequence numbers delivered before, then $i > j_1$. Put differently: the channel may delay a message but no message can be "overtaken" by N or more messages.

  Under this constraint, specify a decision strategy for B that is secure even if A sends an arbitrary number of messages. Hint: have B store the N largest sequence numbers received so far.

Exercise 6.13 (Implement a Simple Peer-to-Peer Ledger) Modify your code from Exercise 4.6 to add the following features:

1. The system still keeps a `Ledger` (see Fig. 4.18).
2. Each client can make `SignedTransactions` (see Fig. 6.5), i.e., what is broadcast is now objects of the type `SignedTransaction`.
3. The sender and receive of a transaction are now RSA public keys encoded as strings. The client can only make a transaction if it knows the secret key corresponding to the sending account. This ensure that only the owner of the account can take money from the account. In a bit more detail, you have to find a way to encode and decode RSA public keys into the `string` type. If we call the encoding of `pk` by the name enc(`pk`), then the amount that "belongs" to `pk` is `Accounts[enc(pk)]`. To transfer money from `pk` one makes a

```
package account

type SignedTransaction struct {
     ID        string // Any string
     From      string // A verification key coded as a string
     To        string // A verification key coded as a string
     Amount    int    // Amount to transfer
     Signature string // Potential signature coded as string
}

func (l *Ledger) SignedTransaction(t *SignedTransaction) {
     l.lock.Lock() ; defer l.lock.Unlock()

     /* We verify that the t.Signature is a valid RSA
      * signature on the rest of the fields in t under
      * the public key t.From.
      */
     validSignature := true

     if validSignature {
             l.Accounts[t.From] -= t.Amount
             l.Accounts[t.To] += t.Amount
     }
}
```

Figure 6.5 The `SignedTransaction` type.

`SignedTransaction` where `pk` is encoded and put in the `From`-field. An encoding of the RSA public key to receive the amount is placed in the `To`-field. All the fields (save `Signature`) are then signed under `pk` (using the corresponding secret key) and the signature is placed in `Signature`. A `SignedTransaction` is valid if the signature is valid. Only valid transactions are executed. The invalid transactions are simply ignored.

Implement as in Exercise 4.6 with these additions:

1. When a transaction is made, broadcast the `SignedTransaction` object.
2. When a transaction is received, update the local `Ledger` object if the `SignedTransaction` is has a valid signature and the amount is non-negative.

Add this to your report:

1. How you TA can easily run your system, how is it started, what kind of commands does it take and so on.
2. Test your system and describe how you tested it.
3. If the test is automated, which is preferable, then describe how the TA can run the test.

You do not have to:

1. Handle overdraft, i.e., we allow that accounts become negative.

2. Protection against cheating parties (neither Byzantine errors nor crash errors).

7

Synchrony (DRAFT)

Contents

7.1 Physical Time

When building distributed systems it is sometimes useful, or even necessary, to have access to the physical time. If for nothing else, it is nice if your calendar system can remind you when it is time for lunch. For the sake of measuring time, most consumer grade computers are equipped with a quartz clock which measures time using an electronic oscillator regulated by a quartz crystal. Typical quartz clocks drift away from the real physical time by about 1 second in about 10 days. This is certainly enough to get you to lunch at a reasonable time, since it would take 50 years to drift half an hour. But note that by the above number a quartz clocks drifts about 2^{-20} seconds per second, and a modern CPU can execute about 1000 instructions in 2^{-20} seconds. For a modern computer, 2^{-20} is a long time, so when judged by modern computers, quartz clocks are horrible at keeping time.

Another alternative is atomic clocks which use the electron transition frequency of atoms for measuring time. These clocks lose about 1 second in 1 million years and are used for defining international standards of time: since we have no more precise way of measuring time, these devices becomes our *de facto* standard for what time is. However, atomic clocks are too bulky and expensive to be put in consumer grade computers.

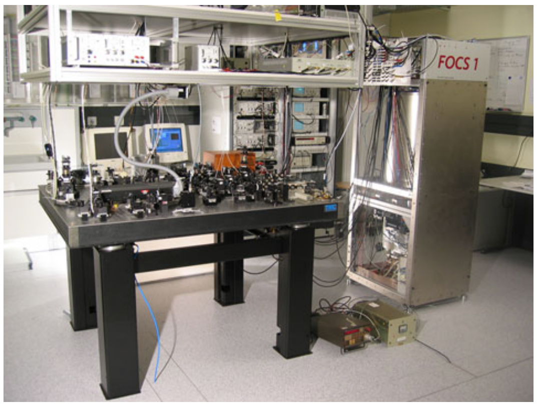


Figure 7.1 FOCS 1, an atomic clock in Switzerland which drifts at most one second in 30 million years

7.2 Clock Synchronisation

When you buy a new watch, you would typically set the time to be the same as some other clock that you believe is correctly showing the current time. Similarly, computing devices can synchronise their clock by interacting with more accurate clocks to which they are connected via a computing network. In practice, this means there are a few organisations owning very expensive and very precise atomic clocks (which provide essentially error-free measures of time), who can help the billions of computing devices equipped with low-accuracy quartz clocks keep their measure of time from drifting too much.

It turns out that clock synchronisation in distributed systems is harder than setting the time on your own watch. In particular, here is a solution that *does not* work: the device with the lousy clock asks the device with the precise clock for its current time, and then sets it as its own. The problem with this solution is clearly that transmitting messages takes time: by the time you receive the time measured by the atomic clock, some more time has passed, so the time you received is not the current time anymore! Therefore, protocols for clock synchronisation are a bit more complicated than the one described above.

Two of the main systems for clock synchronisation are the GPS system and the NTP protocol.

7.2.1 GPS Clock Synchronisation

The global positioning system consists of a large number of satellites in low orbit. Each of them is equipped with an atomic clock and they constantly transmit their position and their time. Their position consists of three spatial coordinates (x, y, z) and the time consist of one time coordinate t. Together this gives a 4-dimensional coordinate (x, y, z, t) which the satellites broadcast publicly and can be received by anyone with a GPS receiver which has the satellite in line of sight.

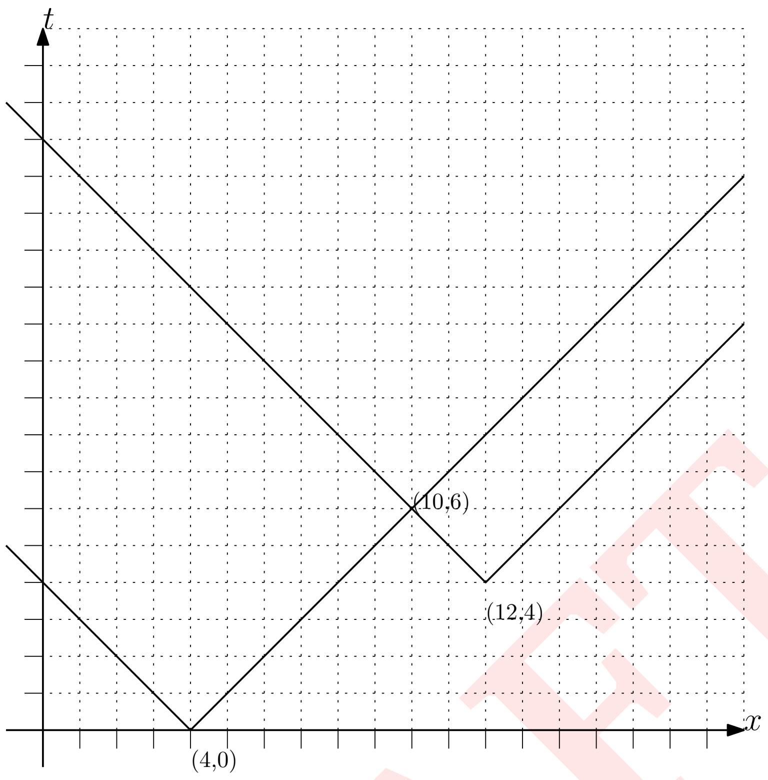


Figure 7.2 GPS in two dimensions. Time runs upwards (unit: 1 second) and space runs to the right (unit: c meters (where c is the speed of light in meters per second)). A satellite at position $x = 4$ and at time $t = 0$ sends the signal $(4, 0)$. When you receive it you can be anywhere on the left-most inverted pyramid. Another satellite at position $x = 12$ and at time $t = 4$ sends the signal $(12, 4)$. When you receive the signal you can be anywhere on the right-most inverted pyramid. But if you receive both signals at the same time, you know that you are at position 10 and that the time is 6. Similarly, for four dimensions you need readings from four satellites.

Notice that if you receive signals (x_1, y_1, z_1, t_1) and (x_2, y_2, z_2, t_2) from two different satellites, the time coordinates t_1 and t_2 might not be the same, since the time it takes for the signal to reach you is a function of your distance to the satellites, and your distance to the two satellites might not be the same. However, it turns out that if you receive four signals $(x_1, y_1, z_1, t_1), \ldots, (x_4, y_4, z_4, t_4)$, then you can compute your own position and time (x_0, y_0, z_0, t_0). Oversimplifying a bit, this corresponds to having a system with four equations and four unknowns, which can then be solved. We will not go into details of the GPS equations here (See Figure 7.2 for a simplified example with a single spacial dimension).

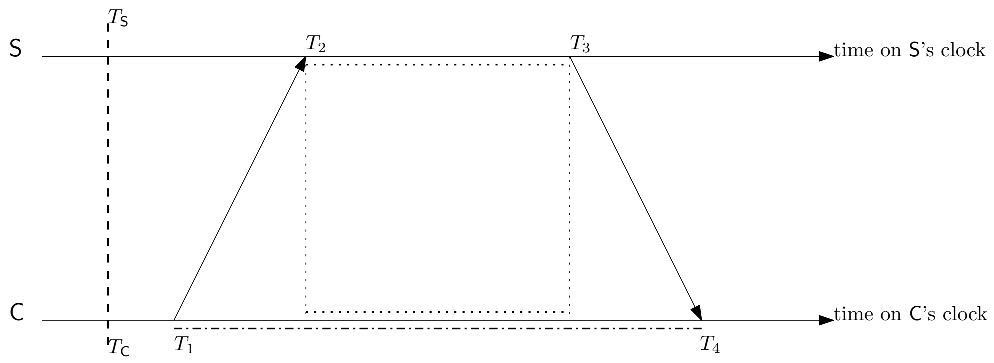


Figure 7.3 The Core of the NTP Protocol illustrated.

7.2.2 NTP Clock Synchronisation

We will instead have a look at the details of the Network Time Protocol. In this protocol we imagine a server S which has an atomic clock, or some other very precise clock. There is a client C with a drifting clock which occasionally needs to be synchronised.

The algorithm works under two assumptions:

- (A1) During the time it takes to run the protocol, the clocks of the server and the client only drift negligibly apart.
- (A2) The time it takes to send a package from client to server is the same as it takes to send from the server to the client.

It is natural to ask whether these assumptions are satisfied in practice. Assume that the server has an atomic clock, that the client has a quartz clock, and the entire protocol takes about a second to terminate. Then the drift of the between the clocks will be about 2^{-20} seconds, which is small enough to be ignored and therefore assumption A1 holds. Assumption A2 is less straightforward: while it might be true that the time it takes to send packets between two devices is approximately the same on average, network errors, packet retransmission and unstable connections can easily negate assumption A2 in practice. We will return to this later on.

Given the two assumptions, we are now ready to describe the main idea behind the NTP protocol (with many details omitted). When a client wants to synchronise its clock, the following protocol in Fig. **??** is executed.

To understand why the protocol works the way it does, consider Fig. **??**. The top vertical line is time measured on the clock of the server. The bottom vertical line is time measured on the clock of the client. Times as measured on the clocks are plotted at the same physical time. Now take the clocks at the physical times indicated by the dashed vertical line to the left. The clock at the server is T_S. The clock at the client is T_C. So, the clock of the client is $\mathsf{Offset} = T_C - T_S$ ahead. We call this the offset of the client's clock (relative to the clock of the server). Notice that with this definition of offset, it holds that $T_S = T_C - \mathsf{Offset}$. In the rest of the

1. C sends a "time request" message to S and stores its current system time T_1.
2. Upon receiving the "time request" message, the server S stores its current system time T_2.
3. The server S prepares a "time response" message, which includes the time T_2. Right before sending the response to the client C, the server S measure its current system time again, call it T_3, and includes it in the message.
4. Upon receiving the message ("time response", T_2, T_3), the client C measures its current system time T_4. It can now compute two numbers:

   $$\mathsf{TransEst} = \frac{(T_4 - T_1) - (T_3 - T_2)}{2}$$

   $$\mathsf{OffsetEst} = (T_1 + \mathsf{TransEst}) - T_2 \ .$$

   The number OffsetEst is an estimate of how much the client's clock is ahead of server's clock. The client adjusts its own clock using this estimate.

Figure 7.4 The core of the NTP protocol. See also Fig. 7.3

text, an offset is always how much the clock is *ahead*. One of the goals of the NTP protocol is to estimate this offset. Assumption A1 can be now rephrased as saying that the offset is more or less constant during the execution of the protocol. In other words, the offset could have been measured by subtracting the clock of the server from the clock of the client at any fixed physical time. Now, let Trans be the time it takes for a message to travel from the client to the server or the other way around (remember that by assumption A2 the two times are equal). Then, at the physical time when the server measures the time T_2, the clock of the client would show the time $T_1 + \mathsf{Trans}$ (this is where the first vertical dotted lines touches the clock line of the client in the picture).

Then the offset could be measured as $\mathsf{Offset} = (T_1 + \mathsf{Trans}) - T_2$. Therefore, to compute the offset, it is enough to be able to estimate the transmission time.

Unfortunately, since the clock of the client and server are not synchronised initially, there is no way to compute the transmission time precisely. For instance, the time it takes to send a message from the client to the server is *not* $T_2 - T_1$ as T_1 is measured on the client's clock and T_2 is measured on the server's clock, and these two clocks are not in sync. The NTP algorithm therefore uses the following trick instead: Notice that $T_4 - T_1$ is the total time it took to run the protocol. This is a well defined measure of time since T_1 and T_4 are measured on the same clock. Similarly, the time spent on processing at the server side is $T_3 - T_2$, which is also a well defined measure of time since it is performed on the same clock. Therefore, the total time it took to send both messages was $(T_4 - T_1) - (T_3 - T_2)$, since clocks do not drift noticeably during the protocol (assumption A1). Under the assumption A2 (that the message transfer time is the same in both directions) we can divide this by 2 to get an estimate of Trans, and hence an estimate of Offset, so this explains the computation done by the client in the protocol.

Adjusting the Clock

Now, if the estimated offset is positive, then the client's clock is ahead and it will turn down the speed of its clock a bit to fall back. If the offset is negative, then the client's clock is behind and it will turn up the speed of its clock a bit to catch up. Notice that the client does *not* just set its clock to the new estimate of what the time is at the server. When it is ahead, this would force time to travel backwards on the clock of the client which will be bad for many processes on the client. But sudden jumps forward in time can be disturbing too. Imagine that you used your computer to time the boiling of an egg for breakfast, and then the clock of the computer suddenly jumps forward by five minutes. When the timer goes off, the egg might not be to your taste. It is therefore better to adjust the speed of the clock a bit. In many cases this is also better at preventing future drifts.

Exercise 7.1 (precise guarantee) Assume that the clocks of the server and client drift at most ϵ during the execution, i.e., the change in their offsets does not change by more than ϵ. Assume that the time it takes to send the message in one direction is no more than δ plus the time it takes to send the message in the other direction. Assume for simplicity that the client right after measuring T_4 subtracts Offset from its clock. Assume that the client updates it's clock by subtracting OffsetEst at the end of the protocol. Give an upper bound on the absolute offset of the client's clock at the end of the protocol.

Revisiting the Assumptions

As already mentioned, the assumption that the transfer time is the same in both direction is a brave one. Even if this is true when the network is in good condition, a burst error affecting only one direction could falsify the assumption. To avoid such a situation NTP runs the core protocol several times and then adopts the execution where TransEst was lowest. The rationale is that when errors occur, then transmission times tend to get higher, so throwing away the executions with the highest execution times will in general also throw away the executions where there were network errors. Even with these added bells and whistles the assumption is somewhat brave. Sometimes there is just a difference in the up link and down link of a computer. The next exercise lets you think a bit about this situation.

Exercise 7.2 (different up and down latencies) Consider a system where the transmission times in the two directions are not the same. For simplicity assume that they are always t_1 respectively t_2. But you do not know these values, you are just guaranteed that whenever you send a message from the client to the server it takes exactly t_1 seconds, and whenever you send a message from the server to the client it takes exactly t_2 seconds. For simplicity assume also that the clock of the server and client do not drift during the execution of the protocol, i.e., assumption A1 holds perfectly. However, you start from a situation where the clocks of the server and the client can be arbitrarily far apart. Can you design a protocol to synchronise the clock of the client with the one of the server? How precise can you make your protocol?

Exercise 7.3 (quorum clock synchronisation I) One big problem with NTP is that the client puts all its trust in the server. If the server is corrupted it can set the clock of the client to something wrong, which might have disastrous consequences for the safety of the client. Assume that you are a client and that you have access to the three clock servers of which you are willing to trust that at least two are honest and meet assumptions A1 and A2 perfectly. Assume that the clock of clocks of the honest servers are perfectly synchronised. Try to design a protocol which mitigates the influence of a Byzantine server.

Exercise 7.4 (quorum clock synchronisation II) This is the same exercise as above, but we consider a more realistic setting. In practice A1 and A2 do not hold perfectly and the clocks of honest servers will not be perfectly the same. We have to account for that in out algorithm. As a first step, assume that A1 and A2 still holds perfectly for the two honest servers. But assume that the clocks of the two honest servers can be a bit away from the real time and that they might be different. Specifically, assume there is some δ such that both clocks of the honest servers are within δ from the real time. Try to design a protocol which mitigates the influence of a Byzantine server. How close to the real time can you guarantee to be? Try to make an algorithm that is allowed to use δ and on which does not use δ. Can you do better when you can assume to know δ?

7.3 Round-Based Protocols

To study the general power of using time in distributed systems it is usual to first consider the fully synchronous round-based model. In the simplest instantiation there are n parties or processes called $\mathsf{P}_1, \ldots, \mathsf{P}_n$. The protocol, denoted by π, proceeds in rounds. The idea is that in each round each process is allowed to send a message to each other process. Sending nothing is an option, which is denoted NoMsg in the following. In this model, we assume that all processes have perfectly synchronised clocks and that transmission time is fixed such that a message sent by a correct process in a certain round will always be available to the receiver when the next round starts.

7.3.1 When a Sent Message Arrives

While this model is simple and easy to work with, it is clear from our previous discussion that the assumptions made are hardly realistic. There is some hope, however. Even if clocks are not perfectly synchronised, we can still hope to set bounds on clock drift and transmission time, and predict when a message ought to arrive. More precisely, suppose a client expects that a server will send a message at time t. Assume that the client knows that the absolute offsets of the server's clock and its own clock relative to real time are at most Offset. Assume also that the client knows that it takes at most Trans seconds to send a message. Then, if nothing was received at time $t + 2\mathsf{Offset} + \mathsf{Trans}$ on the clock of the client it can conclude that the message was not sent. Namely, the server will send the message at time t on its own clock, which is no later than at real time $t+\mathsf{Offset}$. Therefore the message arrives no later than at real time $t + \mathsf{Offset} + \mathsf{Bound}$. At real time $t + \mathsf{Offset} + \mathsf{Bound}$ the clock of the client shows no more than

$$(t + \mathsf{Offset} + \mathsf{Bound}) + \mathsf{Offset} = t + 2\mathsf{Offset} + \mathsf{Trans}\ .$$

7.3.2 Accounting for Computation Time

We now generalise the above idea to a protocol π consisting of many rounds. We will argue that if we know bounds on clock drift, transmission time and computation time of the computations done in each round, we can set timeouts in such a way that we avoid dropping messages that arrive too late. This will imply that we can run a protocol that was designed for the fully synchronised round-based model in the more realistic setting of imperfect clocks and varying transmission times.

For simplicity we will assume that the computation that needs to be done in each round takes about the same time in each round. We use $\mathsf{MaxComp}$ to denote a positive time bound on how long it takes any party to complete the computation of any round once all the inputs are ready. We also assume a positive time bound $\mathsf{MaxTrans}$ on how long it maximally takes to send a message between two correct processes and a positive time bound $\mathsf{MaxDrift}$ on how long any correct process drifts from the real time. Note that we can use clock synchronisation to keep $\mathsf{MaxDrift}$ down.

From these numbers one can compute the slot length:

$$\mathsf{SlotLength} = 2\mathsf{MaxDrift} + \mathsf{MaxTrans} + \mathsf{MaxComp}\ .$$

Note that this is just the time on the clock of the receiver that it would take for a sent message to arrive, but including now also the time it takes to compute the message. By making the rounds this long, in each round each honest party has time to compute the messages to send, send them and have them arrive at the other honest parties in time.

We assume that there is a time t_0 at which the parties agree that they start running the protocol. At this time we also assume that they received their inputs for the protocol. We can assume that the input to each party P_i is (t_0, x_i) and that all honest parties have the same t_0 as input. We assume that (t_0, x_i) arrives before the clock of P_i reach t_0 such that P_i can start running the protocol at (local) time t_0.

Each round runs within a time slot. Rounds are indexed by natural numbers r. Round r begins at (local) time

$$\mathsf{SlotBegin}^r = t_0 + r \cdot \mathsf{SlotLength}\ .$$

We assume that the computation performed by P_i is described by an algorithm

$\mathsf{Compute}$. It takes as input the round number r and the messages received in the previous round. It output the messages to be sent in the next round. Note that P_i also sends a message to itself. It is just stored locally. This message can be used to store the local state of the parties, i.e., the values it needs to remember for future rounds. In round $r = 0$ it only takes the input of P_i as input.

1. Each P_i gets input (t_0, x_i) before local time t_0 and computes $\mathsf{SlotBegin}^r$ for $r = 0, 1, \ldots$.
2. At time $\mathsf{SlotBegin}^0$ party P_i computes $(m^0_{i,1}, \ldots, m^0_{i,n}, \mathsf{State}_{i,0}) = \mathsf{Compute}(0, x_i)$. The local time at P_i is now at most $\mathsf{SlotBegin}^0 + \mathsf{MaxComp}$.
3. Now send $(\textsc{msg}, 0, m^0_{i,j})$ to each P_j. This message arrives at local time at most $\mathsf{SlotBegin}^0 + \mathsf{MaxComp} + 2\mathsf{MaxDrift} + \mathsf{MaxTrans}$ at P_j. And $\mathsf{SlotBegin}^0 + \mathsf{MaxComp} + 2\mathsf{MaxDrift} + \mathsf{MaxTrans} \leq \mathsf{SlotBegin}^1$.
4. In rounds $r = 1, 2, \ldots$, party P_i runs as follows:
   1. Until local time $\mathsf{SlotBegin}^r$, receive and store messages of the form $(\textsc{msg}, r - 1, m^{r-1}_{i,j})$ from P_j.
   2. At time $\mathsf{SlotBegin}^r$, for each P_j where no $m^{r-1}_{j,i}$ is stored, define $m^{r-1}_{j,i} = \textsc{NoMsg}$. Then compute $(m^r_{i,1}, \ldots, m^r_{i,n}) = \mathsf{Compute}(m^{r-1}_{1,i}, \ldots, m^{r-1}_{n,i})$. Now the local time is at most $\mathsf{SlotBegin}^r + \mathsf{MaxComp}$.
   3. Now send $(\textsc{msg}, r, m^r_{i,j})$ to each P_j. This message arrives at local time at most $\mathsf{SlotBegin}^{r-1} + \mathsf{MaxComp} + 2\mathsf{MaxDrift} + \mathsf{MaxTrans}$ at P_j. And $\mathsf{SlotBegin}^{r-1} + \mathsf{MaxComp} + 2\mathsf{MaxDrift} + \mathsf{MaxTrans} \leq \mathsf{SlotBegin}^r$.

Figure 7.5 A generic round-based protocol.

A round-based protocol now proceeds as in Fig. 7.5.

Using Clocks make You Slow

If an honest party sends a message which does not arrive before the timeout (and hence is replaced by NoMsg), then we say that a correct message was dropped by the timeout. Note that if all out assumptions on MaxTrans, MaxDrift, and MaxComp are correct, then it will never happen that a correct message is dropped by the timeout. This follows from the way we compute the slot length.

A warning about the round-based model is in order, however. We argued above that if a process is correct, then its messages will never be dropped by a timeout. Conversely, this means that if a process sends NoMsg, in a round where the protocol says something should be sent, then that process is not correct. In the Byzantine faults model this allows P_i, when it receives NoMsg from P_j, to assume that P_j is Byzantine corrupted. Since our models typically do not guarantee anything about the safety of corrupted parties, it follows that the protocol after this point can ignore the safety of party P_j. It can therefore have devastating consequences to miss a timeout. Therefore all the time bounds have to be put such that even in unfortunate conditions they can be met by all honest processes except with negligible probability. Most computers should be able to synchronise their clocks within a second of real time. The time to deliver a message, however, under worst case conditions over the internet is problematic to set conservatively. Even a bound as high as five seconds may be too daring: recall, it must never happen that it takes this long to deliver a message. Even with these bounds, each round will suffer a penalty of seven seconds. This means that a protocol with a 1000 rounds would suffer a delay of about two days. Since the typical delivery time of a network is in the millisecond range, a protocol that does not care about timeouts could in principle execute 1000 rounds of communication in about a second. So it costs us the difference between a second and a day to work with round-based synchronous protocols as opposed to the asynchronous protocols we will explore in the next chapter. These are only rough estimates, but their purpose is to illustrate the potential penalty of working with explicit time and timeouts in distributed systems.

The takeaway message is that in fully synchronous systems, each round takes the worst-case time to send a message, which might be much, much longer than the typical network delivery time. Therefore synchronous systems tend to be much slower than asynchronous systems.

7.4 Synchronous Round-Based Communication as an Ideal Functionality*

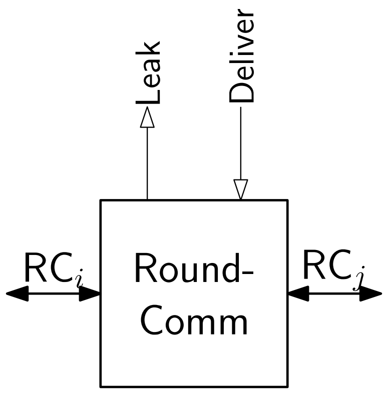
It is possible to capture the idea of synchronous round-based communication as an IF such that it can be used in other contexts and even in combination with other types of synchronous protocols and even asynchronous ones. There are many ways to do this. In Fig. 7.6 we give a fairly simple one for illustration.

It has a variable $\mathsf{ReactionTime}$ which says how long parties have to compute the messages for the current round from when they received the messages from the previous round. It should ideally be set such that no honest parties misses the timeout. If it is set more aggressive, then a missed timeout ($\textsc{NoMsg}$) should not be taken as the sender being corrupted.

The *Init* command allows the parties to specify $\mathsf{ReactionTime}$. If they give different values, the user contract is broken and hence a secure implementation is allowed to behave arbitrarily. This just means that a secure implementation is allowed to assume that the parties agree on the value. By specifying the user contract accordingly an implementation can also assume that the honest parties initialize close to each other in time, for instance within time at most $2\mathsf{MaxDrift}$. They could input $\mathsf{MaxDrift}$ via the initializaiton.

The *Input, Honest, Timely* command says that if an honest party inputs its next message before $\mathsf{ReactionTime}$ time units after its last output, then it is guaranteed to be delivered.

The *Input, Honest, Late* command allows the adversary to set the message of an honest party to be $\textsc{NoMsg}$ if the party was late. Not that it might also choose not to do this and thereby allow a late message to actually be delivered. The reason of this design choice is that it makes it easier to implement the IF. If we had required that $\mathsf{Msg}_{i,j} = \textsc{NoMsg}$ exactly when $t_{\textsc{now}} = \mathsf{LastOutput}_i + \mathsf{ReactionTime}$,



Ports The channel connects n parties named $\mathsf{P}_1, \ldots, \mathsf{P}_n$. For each party P_i it has a port called RC_i and one called Init_i. It has special ports Leak, Send and $\mathsf{Deliver}$ used to model that the network does not hide what is sent and that the adversary (or whomever controls the special ports) has some power over when a message is delivered.

State For each pair of parties $\mathsf{P}_i, \mathsf{P}_j$ and each round r it keeps a variable $\mathsf{Msg}^r_{i,j}$ which is initially set to $\perp$. It keeps for reach P_i a variable $\mathsf{LastOutput}_i$ which is initially $\perp$ and $\mathsf{Round}_i = 1$. It has access to the current physical time via a variable t_{NOW}. It also keeps a variable $\mathsf{ReactionTime}$ initially set to $\perp$.

Init On the first input $\mathsf{ReactionTime}_i$ on Init_i set $\mathsf{ReactionTime} = \mathsf{ReactionTime}_i$ and set $\mathsf{LastOutput}_i = t_{\text{NOW}}$. It is part of the user contract that all parties initialize the IF close to each other in time and that they all input the same value for $\mathsf{ReactionTime}$.

User Contract [Fill In].
If the user contract is broken by the honest parties the IF allows the adversary to specify all outputs as it want. So the below commands are only guaranteed the specified behaviour when the user contract is fulfilled by the honest parties.

Input, Honest, Timely On input (P_j, m) on RC_i, where P_i is honest and $\mathsf{Msg}^{\mathsf{Round}_i}_{i,j} = \perp$, set $\mathsf{Msg}^{\mathsf{Round}_i}_{i,j} = m$ and output $(\mathsf{P}_i, \mathsf{P}_j, m)$ on leak.

Input, Honest, Late On input $(\mathsf{P}_i, \mathsf{P}_j, \text{NoMsg})$ on Send, where P_i is honest and $t_{\text{NOW}} \geq \mathsf{LastOutput}_i + \mathsf{ReactionTime}$ and $\mathsf{Msg}^{\mathsf{Round}_i}_{i,j} = \perp$, set $\mathsf{Msg}^{\mathsf{Round}_i}_{i,j} = \text{NoMsg}$.

Input, Corrupt On input $(\mathsf{P}_i, \mathsf{P}_j, m)$ on Send, where P_i is corrupt set $\mathsf{Msg}^{\mathsf{Round}_i}_{i,j} = m$.

Deliver On input P_i on $\mathsf{Deliver}$, where $\mathsf{Msg}^{\mathsf{Round}_i}_{i,i} \neq \perp$ and $\mathsf{Msg}^{\mathsf{Round}_i}_{j,i} \neq \perp$ for all P_j proceed as follows. Output $(\mathsf{Msg}^{\mathsf{Round}_i}_{1,i}, \ldots, \mathsf{Msg}^{\mathsf{Round}_i}_{n,i})$ on RC_i, set $\mathsf{Round}_i = \mathsf{Round}_i + 1$, and set $\mathsf{LastOutput}_i = t_{\text{NOW}}$.

Figure 7.6 Round-Based Synchronous Communication Ideal Functionality Round-Comm

then a secure implementation would have to enforce this. But then it would have to know t_{NOW} exactly, meaning that we would need perfect clocks, which is impractical. We therefore instead specify that a late message *might* not be delivered.

The *Input, Corrupt* command allows the adversary to choose its messages at any point in time and even change its mind. This allows the adversary to first see the messages of the honest parties and only then choose its own messages. This is called a rushing adversary. This is the worst case, and therefore the correct model from a cryptographic point of view.

The *Deliver* command allows the adversary to deliver the messages to P_i in round r at any point in time after P_i sent all its messages for round r and all messages for P_i for round r were sent. A liveness property for Round-Comm could require that this actually eventually happens. We could also have a timing property saying that it has to happen within some time bound. We have chosen to skip the details of this. So far the property just guarantees that when P_i gets it messages, they include all messages sent by honest parties in a timely manner.

Exercise 7.5 (Round-Based Synchronous as IF) Implement Round-Comm against any number of Byzantine corruptions. You should specify a User Contract for the IF which makes your protocol a secure implementation of the IF. You should for instance specify what it means that the honest parties initialize Round-Comm close to each other in time, but you might need to add other requirements. Be clear about what assumptions your protocols makes on the network it uses.

8

Synchronous Agreement (DRAFT)

Contents

Attack Model *In this chapter we assume that*

1. *There are n parties.*
2. *Up to t parties might be Byzantine corrupted. We will consider several limits on t, such as $t < n$, $t < n/2$ and $t < n/3$.*
3. *The adversary cannot drop messages.*
4. *The adversary cannot inject messages.*
5. *Protocols are in the synchronous round-based model. This implies that:*
   1. *The adversary is allowed to delay messages at most some known amount of time.*
   2. *The parties have access to synchronised clocks.*

8.1 Introduction

In this chapter we study protocols which allow several parties in a distributed system to agree on a decision even in the presence of a Byzantine adversary which tries to make them disagree. In general we are interested in three types of properties when considering a protocol trying to solve an agreement problem.

Agreement All honest parties make the same decision.
Validity The decision that is made must be sensible in some sense.
Termination If all parties starts running the protocol then all honest parties end up making a decision.

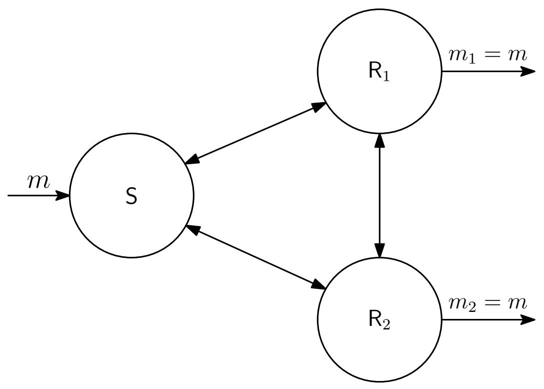


Figure 8.1 A sender sending the same message to two receivers.

As an example consider the setting in Fig. 8.1. There are three parties connected by authenticated channels. There is a sender S which must send the same message m to two receivers R_1 and R_2. We want to tolerate that one of the three parties might be Byzantine corrupted. We want that a corrupted party cannot manipulate the message sent by S. We also want that a corrupted party cannot make R_1 and R_2 output different messages. We could formulate it as follows:

Agreement If both R_1 and R_2 are honest and R_1 outputs m_1 and R_2 outputs m_2, then it always holds that $m_1 = m_2$.[1]

Validity If both S and R_1 are honest, them $m_1 = m$. Similarly, if both S and R_2 are honest, them $m_2 = m$.

Termination If all honest parties start running the protocol, then eventually every honest R_i outputs some m_i.

In this chapter we will look at two agreement problems.

Broadcast The sender S is to send a single message at some agreed time. All receivers eventually output a message or NoMsg and agree on what they output. If the sender S is honest, then only the intended message can be output as coming from S. In particular, if S is honest, then no honest receiver outputs NoMsg.

Byzantine Agreement There are n parties $\mathsf{P}_1, \dots, \mathsf{P}_n$. Each has a single bit b_i as input. They output a common decision d, also a bit. All parties should agree on d. Furthermore, if all honest parties have the same b as input, then $d = b$.

In this chapter we will see some examples of solving these problems. We will see that the two problems are tightly connected. We will also see some lower bounds which explore how well we can expect to solve these problems.

In general agreement problems are very slippery, which is one reason to have

[1] Here it is implicit that it holds even if S is Byzantine corrupted, as the only premise we give is that R_1 and R_2 are honest.

clear definitions of what protocols are supposed to do along with proofs that protocols do what they are supposed to do. Here are some naïve protocols for Broadcast with three parties and one Byzantine corruption that do *not* work.

Example 8.1 S sends m to R_1 and R_2 and R_i outputs $m_i = m$. The problem is that if S is Byzantine corrupted it could send m_1 to R_1 and $m_2 \neq m_1$ to R_1. It is not supposed to do this, but since S is Byzantine corrupted it might choose to deviate from the protocol. So the protocol does not have Agreement. Note that it *does* have Validity and Termination.

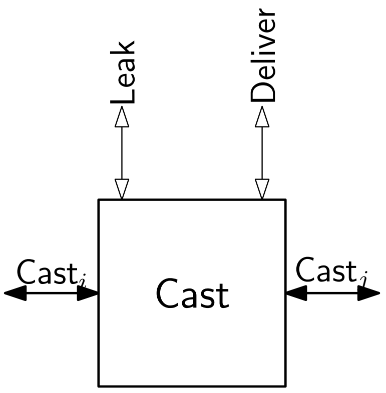
Example 8.2 S sends m to R_1 and R_2. Receiver R_i receives m_i and relays it to the other receiver. Now both receivers have m_1 and m_2. If $m_1 \neq m_2$ then R_i outputs NoMsg. Otherwise R_i outputs m_i. The problem is that if S and R_1 are honest and R_2 is corrupted, then R_2 might chose to send some $m_2 \neq m$ to R_1. Then $m_1 \neq m_2$ at R_1 and hence R_1 outputs NoMsg. But R_1 is supposed to output $m_1 = m$ when S and R_2 are honest. The protocol does not have Validity. Note that it *does* have Agreement and Termination.

Example 8.3 S sends m to R_1 and R_2. Receiver R_i receives m_i and relays it to the other receiver. Now both receivers have m_1 and m_2. If $m_1 \neq m_2$ then R_i outputs nothing. Otherwise R_i outputs m_i. The problem is that if for instance a corrupt S sends m_1 to R_1 and $m_2 \neq m_1$ to R_2, then the receivers never output anything. The protocol does not have Termination. Note that it *does* have Agreement and Validity.

As shown above, getting two out of three of Agreement, Validity and Termination is relatively simple. It is getting all three at the same time that is hard. Sometimes impossible. The first student to present a protocol which always has both Agreement, Validity and Termination with probability 1 for the above setting—with authenticated channels and a single Byzantine corruption—will instantaneously pass the course with an A. Email the solution to your instructor.

Before we progress, we give an example that works for three parties S , R_1 and R_2. It uses signatures, so it does not work with probability 1. The corrupted party can always be lucky to break the signature scheme. But as long as the signature scheme is not broken, the protocol has Agreement, Validity, and Termination.

Example 8.4 We assume that R_1 and R_2 know and agree on the verification key of S. S computes a signature σ on m. Then S sends m, σ to R_1 and R_2. Receiver R_i forwards what it gets to the other receiver. Based on what they received R_i makes a decision as follows. If it has exactly one correctly signed m, then it outputs $m_i = m$. Otherwise it output NoMsg. Agreement: R_1 and R_2 forward what the receive to the other. So if they are both honest, then they make the decision based on the same material, and they use the same rule deterministic to make the decision. Hence they make the same decision. Validity: If S is honest, then it only signs one m, so R_i will receive a unique signed m. So R_i outputs $m_i = m$. Termination: no matter what they receive the receiver output something.



Ports: For each part $\mathsf{P}_i \in \mathbb{P}$ there is a protocol port Cast_i. There are also special ports leak and $\mathsf{Deliver}$.

Syntax: The inputs on Cast_i from parties P_i can be of the form $(\mathsf{bid}, \mathsf{P}_i, m)$, where bid is a broadcast ID and m is a message. Furthermore, parties $\mathsf{P}_j \neq \mathsf{P}_i$ can give inputs of the form $(\mathsf{bid}, \mathsf{P}_i, ?)$ indicating that they know that P_i is about to give an input. Parties P_j can give outputs of the form $(\mathsf{bid}, \mathsf{P}_i, m)$ indicating that they think P_i has broadcast m.

Input: On input $(\mathsf{bid}, \mathsf{P}_i, m)$ on Cast_i the IF outputs $(\mathsf{bid}, \mathsf{P}_i, m)$ on leak.

Output: On input $((\mathsf{bid}, \mathsf{P}_i, m), \mathsf{P}_j)$ on $\mathsf{Deliver}$ the IF outputs $(\mathsf{bid}, \mathsf{P}_i, m)$ on Cast_j.

User Contract: Honest parties must give input according to the following contract.

Synchrony: If in some round any honest party gives an input of the form $(\mathsf{bid}, \mathsf{P}_i, \cdot)$, then in that round all honest parties give an input of the form $(\mathsf{bid}, \mathsf{P}_i,)$.

Unique identifiers: If an honest party is honest then for each bid it gives at most one input of the form $(\mathsf{bid}, \cdot, \cdot)$.

Ideal Functionality Contract: If the inputs are according to the user contract, then the IF gives outputs according to the following rules. It does so by rejecting outputs from the adversary breaking these properties.

IO Input before output: If P_j is honest and some $(\mathsf{bid}, \cdot, \cdot)$ was output on Cast_j, then previously some $(\mathsf{bid}, \cdot, \cdot)$ was input on Cast_j.

Validity: If P_i and P_j are honest and $(\mathsf{bid}, \mathsf{P}_i, m)$ was output on Cast_j, then previously some $(\mathsf{bid}, \mathsf{P}_i, m)$ was input on Cast_i.

Agreement: If P_j and P_k are honest and $(\mathsf{bid}, \mathsf{P}_i, m_j)$ was output on Cast_j and $(\mathsf{bid}, \mathsf{P}_i, m_k)$ was output on Cast_k, then $m_j = m_k$.

Termination:
1. If P_i and P_j are honest and an input of the form $(\mathsf{bid}, \mathsf{P}_i, \cdot)$ is input on Cast_i, then eventually an output of the form $(\mathsf{bid}, \cdot, \cdot)$ is output on Cast_j.
2. If P_j and P_k are honest and an output of the form $(\mathsf{bid}, \mathsf{P}_i, \cdot)$ is given on Cast_j, then eventually an output of the form $(\mathsf{bid}, \mathsf{P}_i, \cdot)$ is given on Cast_k.

Figure 8.2 An ideal functionality Cast for Broadcast.

8.2 Defining Broadcast

The first problem we will consider is the broadcast problem. There are n parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$. One of them sends a message m to each of the other parties which are supposed to receive the same message. For some bound t we want to tolerate that up to t parties are Byzantine corrupted. Loosely speaking the protocol should have these properties:

Agreement If both P_i and P_j are honest and P_i outputs m_i and P_j outputs m_j, then it always holds that $m_i = m_j$.

Validity If both the sender is honest and P_i is honest, them $m_i = m$.

Termination If all honest parties start running the protocol, then eventually every honest P_i outputs some m_i.

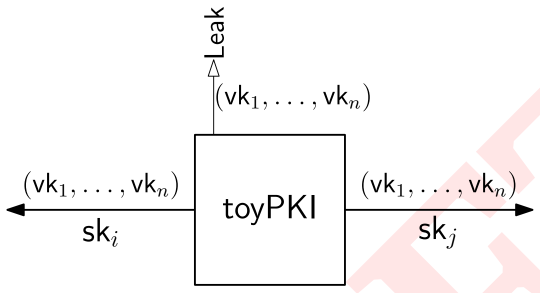
In a more detailed specification of the problem we allow that several messages might be sent and we use a unique broadcast identifier bid to distinguish different broadcasts. Also, we take care of some details like enforcing that honest parties start running in the same round and use each bid at most once. In the round-based model it is important that the honest parties start running in the same round. Otherwise they do not agree on which round they are in and which messages to exchange.

The specification is given in Fig. 8.2. It is slightly informal as we have not defined formally terms like *in the same round* and *eventually*. We leave this out as it is cumbersome to do correctly in the round-based model.

The way we specify Cast might look a bit strange. It is an example of how to under specify an ideal functionality. We want to say that the messages can arrive as they want, except that they have to fulfil certain properties (Agreement, Validity, Termination). We do this by letting the IF offer the adversary to deliver the messages, and then we put the restriction that the adversary must ensure the desired properties. Underspecification is a good thing. If we had been very particular about specifying Cast, like saying P_1 needs to get the message first, then P_2, and so on, then any implementation would need to implement this too, put more restrictions on our protocols. We therefore in general want to make our IF as relaxed as possible. The pattern we used for defining Cast is good for this: Allow it to have *any* behaviour by letting the adversary run the IF, and then put a few restrictions using the desired trace properties. We call this trace-property ideal functionalities. Note that you cannot expect the adversary to follow the restrictions. Adversaries are by definition Byzantine. So you need the IF to enforce the restrictions. For safety properties this is easy: simply ignore instructions that would break the safety property. For liveness properties one have to be a bit more creative. We return to this when we define ACast, the IF for asynchronous broadcast.

It is implicit in the IF Contract that if the inputs are *not* according to the user contract, then the adversary may give any outputs, which makes the IF Cast useless as it might exhibit any behaviour. This is our way to model that an implementation of the IF only has to do something useful as long as the honest parties give inputs as agreed.

The first validity condition makes sure that honest parties have their correct message sent. The second condition ensures that honest parties only have to give outputs if they started running the protocol. The agreement condition ensures that honest parties agree on their outputs. The termination condition ensures that if an honest sender broadcasts a message it is eventually output by all honest receivers. The second condition ensures that if an honest receiver receives a message from a potentially corrupt sender, then eventually all honest parties receive the message from that sender.


Leak
$(\mathsf{vk}_1, \ldots, \mathsf{vk}_n)$
$(\mathsf{vk}_1, \ldots, \mathsf{vk}_n)$
sk_i
toyPKI
$(\mathsf{vk}_1, \ldots, \mathsf{vk}_n)$
sk_j

Ports For all P_i is has a port toyPKI_i. It also has a special port Leak.

Query On any input on any toyPKI_i proceed as follows. If keys were not already generated, then for each P_j generate a random key pair $(\mathsf{vk}_j, \mathsf{sk}_j) \leftarrow G$. Then on leak $(\mathsf{vk}_1, \ldots, \mathsf{vk}_n)$ on Leak. If and when key were generated, output $(\mathsf{vk}_1, \ldots, \mathsf{vk}_n)$ and sk_i on toyPKI.

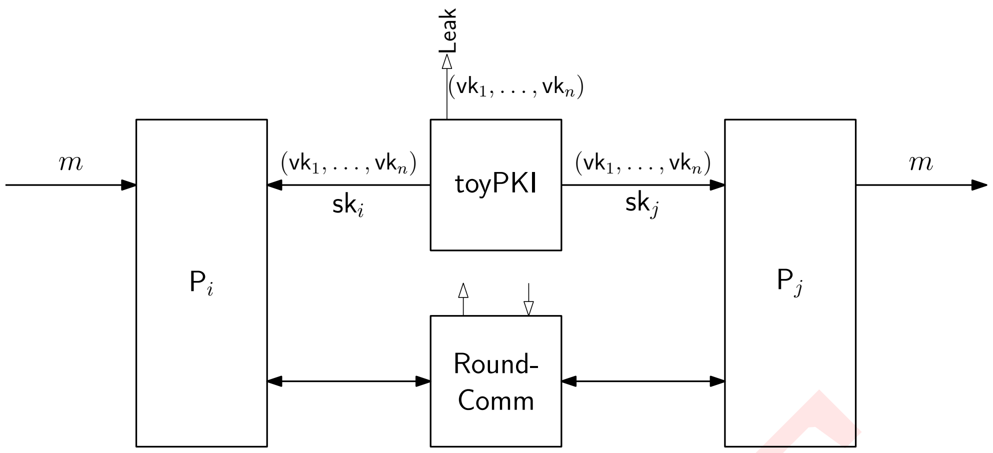
Figure 8.3 The Toy PKI IF toyPKI. It models a setting where each party has a secret signing key for a signature scheme with key generator G and where all parties know and agree on the public verification key of all other parties.

8.3 Broadcast from Signatures

We now look at a solution to the broadcast problem which works for any n and any $t \leq n$. It is a protocol for the synchronous round-based model. It use an IF $\mathsf{Round\text{-}Comm}$ round-based communication. It also uses an IF toyPKI which is a very simple model of a PKI. It is described in Fig. 8.3.

Before we present analysing $\mathsf{DolevStrong}$, let us give an example of a protocol which does not work to motivate why Dolev-Strong is as complicated as it is. The example also opens up the eyes for some of the subtleties in designing secure agreement protocols.

Example 8.5 Assume that we have $n = 4$ parties and that up to $t = 2$ of them can be Byzantine corrupted. Let us sat that S is the sender and that there are


Leak
$(\mathsf{vk}_1, \ldots, \mathsf{vk}_n)$
m
$(\mathsf{vk}_1, \ldots, \mathsf{vk}_n)$
sk_i
toyPKI
$(\mathsf{vk}_1, \ldots, \mathsf{vk}_n)$
sk_j
m
P_i
P_j
Round-Comm

Initialize P_i: When activated the first time query toyPKI and learn the private key sk_i and the public keys of all parties, $(\mathsf{vk}_1, \ldots, \mathsf{vk}_n)$. Then initialize a map $\mathsf{Relayed}_i$ which is $\mathsf{Relayed}_i(\mathsf{bid}, m) = \perp$ for all possible bid and messages m.

Broadcast On input $(\mathsf{bid}, \mathsf{P}_i, m)$ on Cast_i compute $\sigma_i \leftarrow \mathsf{Sig}_{\mathsf{sk}_i}(\mathsf{bid}, m)$, $\mathsf{SigSet} = \{\sigma_i\}$, set $\mathsf{Relayed}_i(\mathsf{bid}, m) = \top$, and send $(\mathsf{bid}, \mathsf{P}_i, m, \mathsf{SigSet})$ to all parties.

Relay In round r after input $(\mathsf{bid}, \mathsf{P}_i, ?)$, if $\mathsf{P}_j \neq \mathsf{P}_i$ receives a message of form $(\mathsf{bid}, m, \mathsf{SigSet})$, where SigSet is a set of signatures and if $\mathsf{Relayed}_i(m) = \perp$, proceed as follows. Call SigSet valid for $(\mathsf{bid}, \mathsf{P}_i, m)$ in round r if it contains signatures σ_k from $r-1$ distinct parties P_k such that $\mathsf{Ver}_{\mathsf{vk}_k}(\mathsf{bid}, m, \sigma_k) = \top$. Furthermore, one of these parties has to be P_i and none of them are from P_j. If SigSet is valid, then compute $\sigma_j \leftarrow \mathsf{Sig}_{\mathsf{sk}_j}(\mathsf{bid}, m)$, let $\mathsf{SigSet}' \leftarrow \mathsf{SigSet} \cup \{\sigma_j\}$ and send $(\mathsf{bid}, m, \mathsf{SigSet}')$ to all parties. Then set $\mathsf{Relayed}_i(\mathsf{bid}, m) = \top$.

Output In round $n + 2$ after input $(\mathsf{bid}, \mathsf{P}_i, ?)$, party P_j computes its output as follows. If there is exactly one message m such that $\mathsf{Relayed}_i(\mathsf{bid}, m) = \top$, then output $(\mathsf{bid}, \mathsf{P}_i, m)$ on Cast_j. Otherwise, output $(\mathsf{bid}, \mathsf{P}_i, \textsc{NoMsg})$ on Cast_j.

Figure 8.4 The Dolev-Strong Protocol, DolevStrong

three receivers $\mathsf{R}_1, \mathsf{R}_2, \mathsf{R}_3$. We assume that R_1, R_2 and R_3 know and agree on the verification key of S. S computes a signature σ on m. Then S sends m, σ to all R_i. Receiver R_i forwards what it gets to the other receivers. Based on what they received R_i makes a decision as follows. If it has exactly one correctly signed m, then it outputs $m_i = m$. Otherwise it outputs $\textsc{NoMsg}$. We can argue Validity and Termination as follows: Validity: If S is honest, then it only signs one m, so R_i will receive a unique signed m. So R_i outputs $m_i = m$. Termination: no matter what they receive the receivers output something. Let us then try to argue Agreement: R_1, R_2, and R_3 forward what the receive to the others. So if they are both honest, then they make the decision based on the same material, and they use the same rule deterministic to make the decision. Hence they make the same decision. This argument does not work. The protocol actually does not have agreement. Can you see what is wrong with this argument? What is wrong in the argument is that claim that the receivers make their decision based on the same material. An attack could work as follows. Assume that S and R_1 are corrupted. In the first round S only sends m, σ to R_1. In round 2 R_1 then forwards m, σ only to R_2. Now R_2 has m, σ and outputs m. But R_3 has nothing and outputs NoMsg. This break agreement.

To fix the above protocol we would need to introduce yet another round where parties forward what they received in the previous round. Then R_3 would learn that R_2 learned m, σ in the previous round. But then the attacker can just introduce the confusion in the next last round. To catch this we would have to introduce yet another round. It seems that this way we then and up with an infinite number of rounds. The trick that makes Dolev-Strong work is one that allows to bound the number of rounds.

Consider the following modification to the above example. When P_2 forwards m, σ to P_3 we require that P_2 adds its own signature on m. So it forwards m, σ, σ_2. And P_3 will in round 2 only accept a forwarded message m if it is signed by both S and one other party. When getting a valid forward in round 2 party P_3 it self forwards and adds its own signature, so it would send m, σ, σ_2. In general, in round r a party only accepts a forwarded message m if it it signed by a total of r different parties including the sender. If so, then its adds its own signature and forwards. This will clearly stop at some point as there are only n parties in the system so no m can be signed by more than n parties. Hence by round $n+1$ all parties will stop accepting incoming messages as they do not have enough signatures attacked. It has Validity as only messages signed by the sender are accepted. It is agreement as the parties accept the same set of messages.

The full Dolev-Strong protocol is described in Fig. 8.4.

Theorem 8.6 *If the signature scheme used by* toyPKI *is unforgeable under chosen message attack, then the protocol* DolevStrong *securely implements* Cast *against any number of Byzantine corruptions.*

Proof We have to prove that DolevStrong behaves as Cast for any number of corruptions. The case with $t = n$ as Cast can behave completely arbitrary when all parties are corrupted, so any behaviour by the protocol is allowed. Assume then that $t < n$ parties are corrupted. We first assume that the protocol is only run once. Let bid be the broadcast ID for which it is run and let P_i be the sender. Below we assume that signature scheme is secure. So if anyone sees a signature σ from P_i on a message M then P_i actually signed M at some point in the past. Below we also assume that all inputs were according to the user contract. If they were not, there is nothing to prove.

Validity

When P_j is honest it only outputs $(\mathsf{bid}, \mathsf{P}_i, m')$ if it saw a signature on $(\mathsf{bid}, \mathsf{P}_i, m)$ from P_i. When P_i is honest it only signs $(\mathsf{bid}, \mathsf{P}_i, m)$ for the correct m.

IO

When P_j is honest it does not output any $(\mathsf{bid}, \mathsf{P}_i, \cdot)$ until $n{+}2$ rounds after seeing an input $(\mathsf{bid}, \mathsf{P}_i, \cdot)$.

Agreement

Assume that P_j and P_k are honest and output $(\mathsf{bid}, \mathsf{P}_i, m_j)$ and $(\mathsf{bid}, \mathsf{P}_i, m_k)$. We need to argue that $m_j = m_k$. It is enough to argue that at round $n{+}2$ it holds that $\mathsf{Relayed}_j(\mathsf{bid}, m) = \mathsf{Relayed}_k(\mathsf{bid}, m)$ for all m, as the parties make their decisions based on for which m it holds that $\mathsf{Relayed}_{\cdot}(\mathsf{bid}, m) = \top$ and they make their decision in round $n+2$. So assume that $\mathsf{Relayed}_j(\mathsf{bid}, m) = \top$. We prove that then $\mathsf{Relayed}_k(\mathsf{bid}, m) = \top$. If $\mathsf{Relayed}_j(\mathsf{bid}, m) = \top$, then P_j in some round received $(\mathsf{bid}, \mathsf{P}_i, m, \mathsf{SigSet})$ which is valid for round r. This means that there is no signature in SigSet from P_j. Therefore $\mathsf{SigSet}' = \mathsf{SigSet} \cup \{\sigma_j\}$ has one more signature from a unique party and hence will be valid for round $r + 1$ for all parties that do not have a signature in SigSet. So, if there is no signature from P_k in SigSet, then $(\mathsf{bid}, \mathsf{P}_i, m, \mathsf{SigSet}')$ is valid for round $r{+}1$ at P_k. And since P_j sends it in round r it will arrive at P_k at round $r{+}1$. This will make P_k set $\mathsf{Relayed}_k(\mathsf{bid}, m) = \top$ in round $r{+}1$. And, if there already *is* a signature from P_k in SigSet, then $\mathsf{Relayed}_k(\mathsf{bid}, m) = \top$ already. This can be seen by inspection of the protocol: the only place where P_k produces signatures on $(\mathsf{bid}, \mathsf{P}_i, m)$ is in Relay, where it also sets $\mathsf{Relayed}_k(\mathsf{bid}, m) = \top$. This ends the proof that if $\mathsf{Relayed}_j(\mathsf{bid}, m) = \top$, then $\mathsf{Relayed}_k(\mathsf{bid}, m) = \top$. By a symmetric argument we can prove that if $\mathsf{Relayed}_k(\mathsf{bid}, m) = \top$, then $\mathsf{Relayed}_j(\mathsf{bid}, m) = \top$. Then proves that if $\mathsf{Relayed}_j(\mathsf{bid}, m) = \mathsf{Relayed}_j(\mathsf{bid}, m)$ for all m. This is what we had to prove.

Termination

Trivial as the protocol for by design stops by round $n + 2$ after it begun.

So far we assumed the protocol was run for only one bid. The above arguments easily extends to the case of multiple runs of the protocol. The reason is that all messages used in an execution with broadcast ID bid contains bid as a prefix. Therefore they cannot interfere with an execution with broadcast ID $\mathsf{bid}' \neq \mathsf{bid}$. □

Exercise 8.1 (Message complexity of Dolev-Strong I) Consider a run of $\mathsf{DolevStrong}$ where all n parties are honest. How many messages are sent?

Exercise 8.2 (Message complexity of Dolev-Strong II) In $\mathsf{DolevStrong}$, if the sender is Byzantine corrupted it can force the honest parties to send a large number of messages by sending signed (bid, m) for a large number of different m. Make an improved version of the protocol where you limit how many messages the honest parties can be forced to send. Argue that the protocol is still secure.

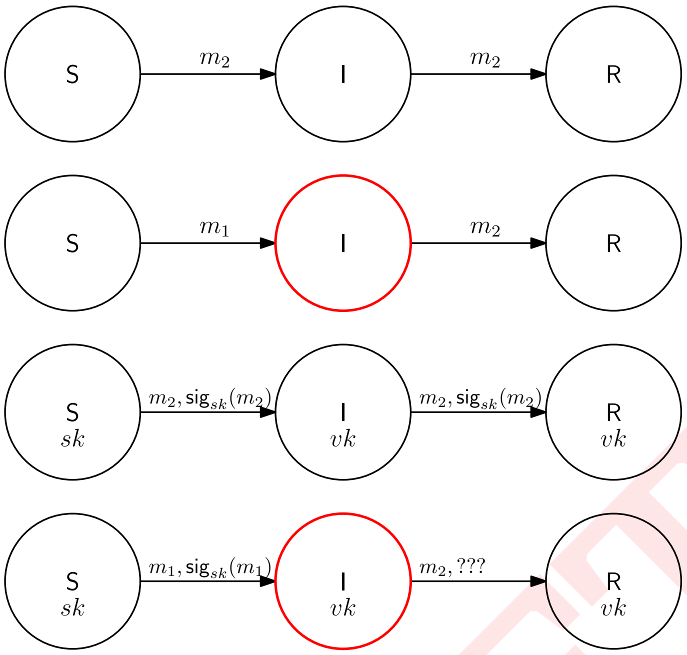

S
m_2
I
m_2
R
S
m_1
I
m_2
R
S
sk
$m_2, \mathsf{sig}_{sk}(m_2)$
I
vk
$m_2, \mathsf{sig}_{sk}(m_2)$
R
vk
S
sk
$m_1, \mathsf{sig}_{sk}(m_1)$
I
vk
$m_2, ???$
R
vk

Figure 8.5 The transferability problem. In the to row a sender S sends a message m_2 via an intermediary I to a received R. In the second row we see the problem that the intermediary can simply change the message, as there is no signature on it. In the third row we see the protocol where there is a signature on the message. The intermediary and the receiver both know the verification key, but only the signer knows the signing key. Therefore, if as in the last row, the intermediary tries to change the message it cannot compute the correct signature to send along.

8.4 Broadcast from Authenticated Channels

In this section we describe how to implement Broadcast given only authenticated channels.

This makes sense in practice as it is often easier to establish pairwise authenticated channels than it is to establish an agreed upon PKI. Implementing Broadcast from only authenticated channels is also an excellent didactic tool. It allows us to dive into how quorums behave in distributed system. Finally it also drives home the difference between the properties we get from authenticated channel (which can be made using symmetric cryptography) and the properties we get from signatures.

In the above section we saw that the Dolev-Strong protocol can achieve Broadcast for any $t \leq n$ Byzantine corruptions, so why are we not done? Well, Dolev-Strong uses signatures, and we now want to do it using only authenticated channels! The property that signatures give us that authenticated channels do not is transferability: if I received a signed message m from you, I can prove it to the rest of the world by sending along the signature. What we will do then is to show

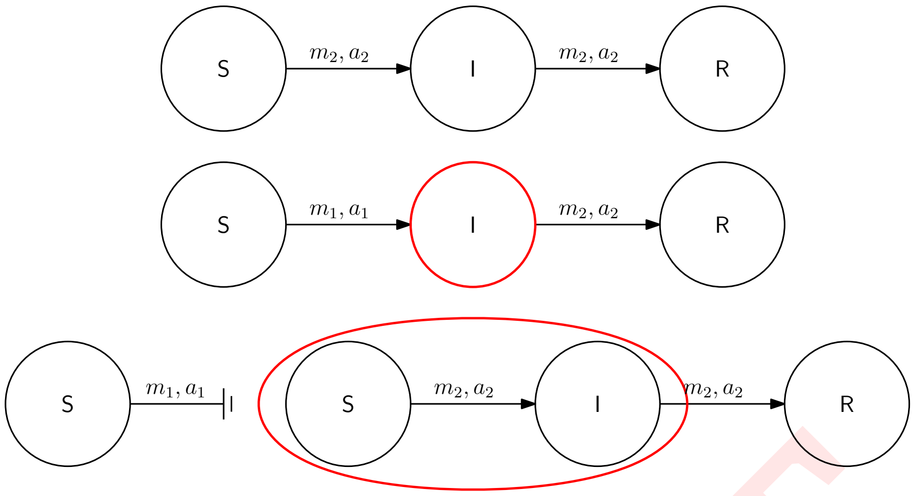


Figure 8.6 The transferability problem cannot be solved without signatures when there is only a single intermediary. No matter what auxiliary information that a protocol with send along with the message as in the top row, the intermediary can always change the message and compute the correct auxiliary information to send along, as illustrated in the second row. The general attack runs as in the bottom row: the corrupted intermediary will simply make a copy of the honest S and run it with a copy of the honest I. Whatever message this makes I send to R, the corrupted intermediary can just send to the real R. From the point of view of R it looks like a correct run, so it must accept m_2.

how to emulate this transferability property of signatures using only authenticated channels. For this emulation to woirk we need that at most $t < n/3$ parties are Byzantine corrupted.

We start by illustrating some of the problems we run into in the model without signatures and the kind of solutions we will use to solve them. In Fig. 8.5, Fig. 8.6, and Fig. 8.7 we illustrate that the problem of forwarding a message securely via one intermediary is impossible. In Fig. 8.9 we illustrate that it is also impossible via two intermediaries. However, it turns out that you can solve it with three intermediaries. You can even solve it in such a way that multiple receivers are guaranteed to get the same message, even when there is one Byzantine corruption. They will simply take majority of the messages receiver. If all the messages received are corrupted, they can conclude that it is the sender that is corrupted, and then they can just use the messages sent via I_1.

The basic idea behind the protocol is pretty simple. Assume that P_S wants to send m to P_R such that P_R can later convince others that it received m. Here S stands for sender and R stands for receiver. The first attempt, which does not quite work, is that P_R asks P_S to send m via all the other parties. So for each P_I (I for intermediary) P_S sends m to P_I and then P_I sends m to P_R. Then P_R accepts m only if it receives m from all P_I. Now clearly P_R can later prove to

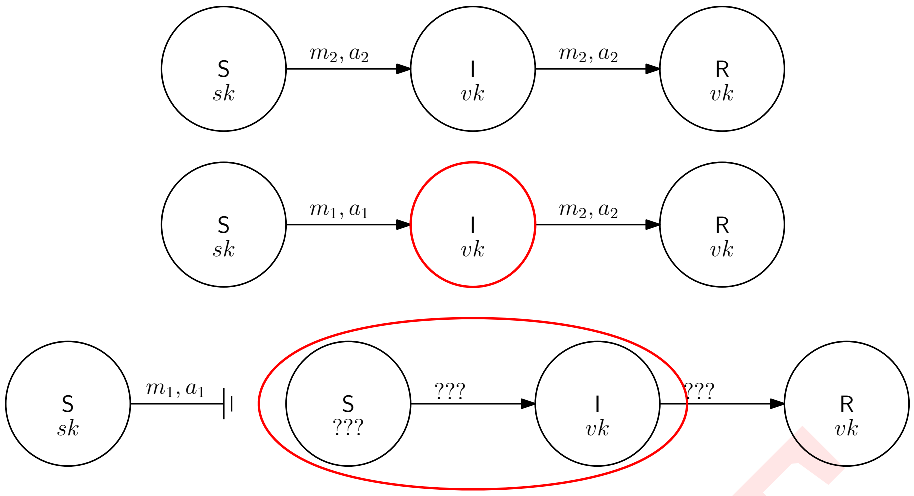


Figure 8.7 The reason why the attack in Fig. 8.6 does not work in the model with signatures is that S has a secret (namely *sk*) which is not known by I. Therefore a corrupted I cannot just make a copy of S,

any other party which message m it received from P_S, as they saw the message already.

The reason it is not that simple is that up to t parties could be corrupted, so t of the parties P_I could forward a wrong message. And there is no way to see if P_S sent a wrong m' to P_I or whether P_I changed the message it got from P_S: recall, we are not allowed to use signatures. We will therefore say the P_R accepts the message m if it received the same m from $n - t$ parties P_I (for simplicity of counting we let P_S and P_R act as intermediaries too). Now corrupted parties cannot prevent an honest P_S from sending m to and honest P_R. But notice now that a corrupted P_S might actually send a wrong m' to some honest P_I and still have P_R accept the message. So now forwarding is a bit harder.

Let us look at how to forward. Assume that P_R accepted m and wants to show m to some third party P_T. When P_R accepted m it must have gotten it from $n - t$ parties. Of these $n - t$ parties, at least $(n - t) - t$ parties are honest. And when $t < n/3$, then $(n - t) - t \geq t + 1$. So if P_R asks everybody to tell P_T that they saw m from P_S, then at least $t + 1$ parties will do so. Look at this from the point of view of P_T: it hears $t + 1$ parties confirm that they saw m from P_S. But there could be t corrupted parties. But at least it knows that $(t + 1) - t = 1$ honest parties confirmed to have seen m. This honest party would only confirm if it actually saw m. Therefore P_S must have sent m. So P_T will accept m. This is how we forward.

There is one subtlety left. Note that P_R could forward to P_T because P_R received the message m from $n - t$ parties. But in the forward P_T accepted the message m after hearing from only $n - 2t$ parties. Therefore P_T cannot *re*forward the message m. For this to be possible P_T needs to hear m from $n - t$ parties. If we want P_T

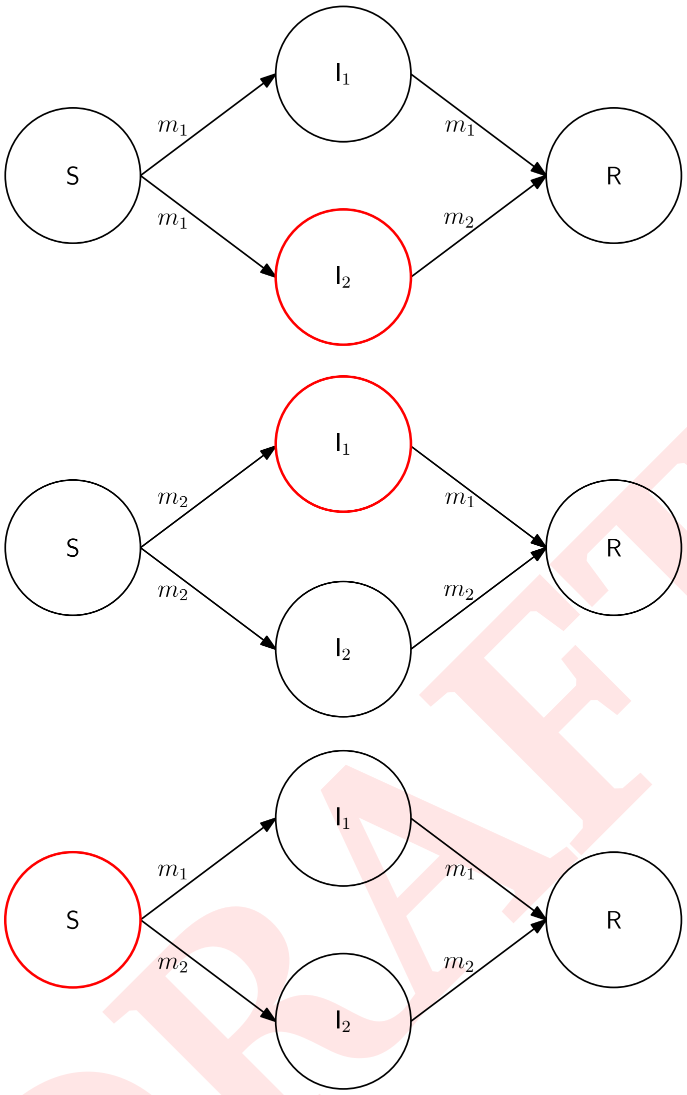


Figure 8.8 Sending via two intermediaries does not work either. In the top row the sender tries to send m_1 and the second intermediary changes the message to m_2. In the second row the sender tries to send m_2 and the first intermediary changes the message to m_1. In both rows the receiver sees the same, so it will make the same decision in both and therefore cannot make the right decision in both. It cannot even conclude the an intermediary cheated. In the bottom row we see the same view created by a corrupted S.

to get a reforwardable message it can be done as follows. First P_R uses the above trick to prove to each P_I that P_S sent m. Then it asks all parties to tell P_T that P_R proved to them that P_S sent m. All $n - t$ honest P_I will do so. So now P_T hears from $n - t$ parties, and thus P_T can now reforward if it needs to.

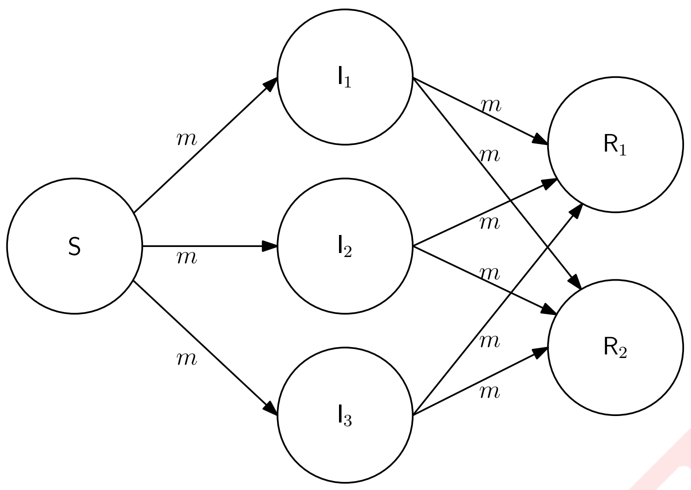


Figure 8.9 The transferability problem can be solved using three intermediaries as long as at most one is corrupted. Here the protocol is illustrated with two receivers. The sender is supposed to send the same message to all intermediaries which just relay them to all receivers. The receivers take majority of the values received. If all three values are different, they output the one sent via I_1.

In more detail the protocol proceeds as in Fig. 8.12. We argue three simple properties of the system.

If $\mathsf{SentBy}(\mathsf{P}_i, m) = \top$ at an honest party, then P_i sent m. This is clearly the case if $\mathsf{SentBy}(\mathsf{P}_i, m) = \top$ was set in *Sent I*. If $\mathsf{SentBy}(\mathsf{P}_i, m) = \top$ was set in *Sent II*, then $|\mathsf{HasSeen}(\mathsf{P}_i, m)| \geq t+1$ at an honest party, so an honest party sent (P_i, m), who must have done so in *Transfer II* because $\mathsf{SentBy}(\mathsf{P}_i, m) = \top$. Now do an inductive argument.

If $\mathsf{SignedBy}(\mathsf{P}_i, m) = \top$ at an honest parties, then P_i sent m. This is trivial as $\mathsf{SignedBy}(\mathsf{P}_i, m) = \top$ implies that $\mathsf{SentBy}(\mathsf{P}_i, m) = \top$.

If $\mathsf{SignedBy}(\mathsf{P}_i, m) = \top$ at an honest P_j that runs *Transfer I*, then within two rounds $\mathsf{SignedBy}(\mathsf{P}_i, m) = \top$ at all honest parties. When $\mathsf{SignedBy}(\mathsf{P}_i, m) = \top$, then $|\mathsf{HasSeen}(\mathsf{P}_i, m)| \geq n-t$, so $t+1$ honest parties have sent (P_i, m). When they did this, they sent the value to all honest parties. So $|\mathsf{HasSeen}(\mathsf{P}_i, m)| \geq t+1$ at all honest parties. So $\mathsf{SentBy}(\mathsf{P}_i, m) = \top$ at all honest parties. So when they receive $(\textsc{Transfer}, \mathsf{P}_i, m)$ all honest parties will send (P_i, m) to all parties because of *Transfer II*. Hence $|\mathsf{HasSeen}(\mathsf{P}_i, m)| \geq n-t$ at all honest by the next round, so all honest set $\mathsf{SignedBy}(\mathsf{P}_i, m) = \top$.

We can now run the Dolev-Strong protocol on top of authenticated channels. Whenever P_i is to sign a message use the *Sign*-command. Whenever P_i is to relay a signed message, use the *Transfer*-command. We could make this more formal by creating an ideal functionality TAC capturing transferable authenticated channels and show that the above channel implements it. Then one would show how to

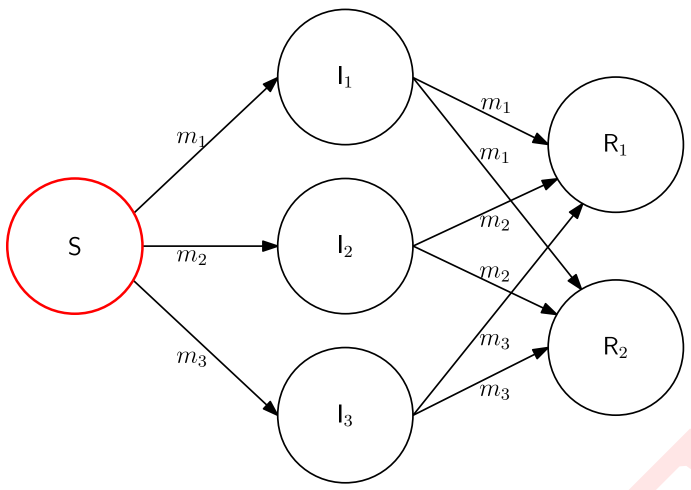


Figure 8.10 Sending a message via three intermediaries, Corrupted sender. If the sender is corrupted it might send three different messages. But when the sender is corrupted, then all intermediaries are honest, as we have assumed that at most one party is corrupted. Therefore they all forward the message as they should. Therefore all receivers see the same messages and make the same decision.

run DolevStrong given TAC instead of signatures. This is a fairly straightforward exercise which does not add any new understanding, so we will not do this here.

8.5 Byzantine Agreement from Broadcast

In our next study of the round-based synchronous model we will look at Byzantine Agreement (BA). In an agreement protocol each party has a vote and the goal is to end up with a common, meaningful decision.

In more detail, a Byzantine Agreement is a protocol between n parties, which as before are denoted as $\mathsf{P}_1, \ldots, \mathsf{P}_n$. The input of P_i is a vote $v_i \in \{0,1\}$. The output of P_i is a result $r_i \in \{0,1\}$. At the beginning of the protocol we set $r_i = \bot$ for all correct processes. When P_i is ready to give an output it sets r_i to 0 or 1 and then never changes it again. We say that the protocol was executed correctly if all correct processes started running the protocol in the same round. We impose the following properties on correctly executed protocols:

Validity: It holds for each correct P_i that if $r_i \neq \bot$ then there exists a correct process P_j such that $r_i = v_j$.

Agreement: It holds for each pair P_i and P_j of correct processes that if $r_i \neq \bot$ and $r_j \neq \bot$, then $r_i = r_j$.

Termination: There exists some constant c such that whenever the protocol starts executing in round t_0, then by round $t_0 + c$ it holds for all correct P_i that $r_i \neq \bot$.

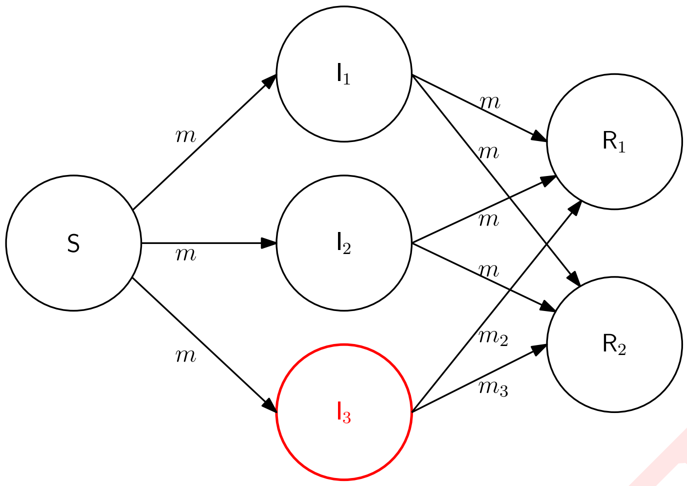


Figure 8.11 Sending a message via three intermediaries, Corrupted intermediary. If an intermediary is corrupted then the sender is honest, as we have assumed that at most one party is corrupted. Therefore S will send the same m twice to all receivers via the two honest intermediaries. Therefore all receivers will output m as they use the majority rule.

Init Initially for all P_i and m let $\mathsf{HasSeen}(\mathsf{P}_i, m)$ be an empty set, let $\mathsf{SignedBy}(\mathsf{P}_i, m) = \bot$, and let $\mathsf{SentBy}(\mathsf{P}_i, m) = \bot$. The set $\mathsf{HasSeen}(\mathsf{P}_i, m)$ is the set of parties claiming to have seen (P_i, m). A party sets $\mathsf{SentBy}(\mathsf{P}_i, m) = \top$ if and only if it is sure party P_i sent m: either it saw P_i send it or it heard $t + 1$ parties claim it. A party sets $\mathsf{SignedBy}(\mathsf{P}_i, m) = \top$ if and only if it is sure it can prove that P_i sent m: it heard at least $2t + 1$ parties claim that P_i sent m.

Sign If P_i wants to sign m it sends (P_i, m) to all parties P_j and sets $\mathsf{SignedBy}(\mathsf{P}_i, m) = \top$.

See If receiving (P_i, m) from any P_j add P_j to $\mathsf{HasSeen}(\mathsf{P}_i, m)$.

Sent I If P_j receives (P_i, m) from P_i it sets $\mathsf{SentBy}(\mathsf{P}_i, m) = \top$.

Sent II If $|\mathsf{HasSeen}(\mathsf{P}_i, m)| \geq t + 1$ set $\mathsf{SentBy}(\mathsf{P}_i, m) = \top$.

Signed If $|\mathsf{HasSeen}(\mathsf{P}_i, m)| \geq n - t$ set $\mathsf{SignedBy}(\mathsf{P}_i, m) = \top$.

Transfer I If $\mathsf{SignedBy}(\mathsf{P}_i, m) = \top$, then P_j can transfer it by sending $(\textsc{Transfer}, \mathsf{P}_i, m)$ to all other parties.

Transfer II On message $(\textsc{Transfer}, \mathsf{P}_i, m)$, where $\mathsf{SentBy}(\mathsf{P}_i, m) = \top$, send (P_i, m) to all parties.

Figure 8.12 A protocol EmuSig emulating signatures given authenticated channels.

Termination says that all correct processes eventually compute a result. Agreement just says that all correct processes will agree on the result they compute. Validity says that the result must be a value that was voted by at least one correct process (this is the part making the result meaningful). In particular, if all correct processes vote the same value, then all correct processes must eventually agree on this value.

We could formulate the above as an ideal functionality BA with BA identifier and so forth, in line what we did with Cast. But this the point we want to drive home in this section is fairly simple, we save a bit on the formalism and just state the BA problem informally as above.

Notice that if there are n processes and up to t Byzantine processes, then Byzantine agreement is impossible if $t \geq n/2$. This is for purely definitional reasons and does not involve an actual attack. To see this assume that $n = 2t$. Consider an execution where $\mathsf{P}_1, \ldots, \mathsf{P}_t$ have input 0 and $\mathsf{P}_{t+1}, \ldots, \mathsf{P}_n$ have input 1. Assume that half the parties are Byzantine corrupted but follow the protocol honestly. By termination and agreement the parties will eventually agree on some output b. Assume now that honest parties were $\mathsf{P}_1, \ldots, \mathsf{P}_t$. Then validity requires that $b = 0$. But if instead the honest parties were $\mathsf{P}_{t+1}, \ldots, \mathsf{P}_n$, then validity requires that $b = 1$. Since all parties follow the protocol honestly there is no algorithmic way to determine who is corrupted, so the algorithm cannot determine which case it is in. So whatever it outputs, it will always be wrong in one of the cases. When we work with Byzantine agreement we therefore always assume that $t < n/2$ of the processes are Byzantine corrupted.

So BA is impossible $t \geq n/2$. On the other hand, it is possible whenever $t < n/2$ if we have access to a Broadcast IF Cast, which is secure for the same t. We simply run n instances of the protocol, where each P_i broadcasts their vote v_i. This ensures that all parties learn a vector $(v_1, \ldots, v_n)$ of votes that they all agree on. If some $v_i = \textsc{NoMsg}$, then simply let $v_i = 0$. Then, take the value v which occurs at least $t + 1$ times in this vector and let the result be $r_i = v$.

Termination follows from termination of the underlying broadcast protocol. Agreement follows from agreement of the broadcast protocol: the parties will agree on $(v_1, \ldots, v_n)$ and will therefore compute the same output. Let us then argue validity. Assume that there are two correct processes, one with $v_i = 0$ and one with input $v_j = 1$. Then validity is trivially fulfilled as all parties output 0 or 1. Assume then that there is a value v such that $v_i = v$ for all correct P_i. Then by validity of the broadcast protocol the value v occurs $n - t$ times $(v_1, \ldots, v_n)$ as there are at lest $n - t$ honest parties and they all have input v. Since $n - t \geq t + 1$ it follows that all honest parties take their output to be $r_i = v$.

8.6 Broadcast from Byzantine Agreement

In this section we argue that one can implement Broadcast given access to an IF BA for Byzantine agreement. The protocol is very simple, so we will skip most of the details. Assume that a sender wants to broadcast a bit b to n parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$. It sends b to all parties. Let b_i be the bit received by P_i. Then $\mathsf{P}_1, \ldots, \mathsf{P}_n$ run BA on $b_1, \ldots, b_n$. Let the result be r. All parties output r. Agreement of Broadcast trivially follows from Agreement on BA. Termination of Broadcast trivially follows from Termination on BA. Validity of Broadcast follows from

Validity of BA. Namely, if the sender is honest it sends b to all honest parties, so all honest P_i have input $b_i = b$. By Validity of BA to result r must be the input for an honest party, so $r = b_i = b$. This allows to broadcast single bit, but a sender wants to send a longer message it can just send it bit by bit.

A disadvantage of the above protocol is that if a sender wants to send a 1000-bit message it would involve running 1000 instances of a BA, which would be very expensive. The next exercise ask you to do better.

Exercise 8.3 (Better Broadcast from BA, Signatures) Show how to broadcast a 1000-bit message using only a single BA. Assume that you have access to toyPKI and make the protocol secure against $t < n/2$ Byzantine corruptions. The protocol should be of the following general form: In round 1 the sender signs m and sends it to all parties. In round 2 all parties forward what they received so far to all other parties. In round 3 all parties forward what they received so far to all other parties. In round 4 the parties run one BA. If the result is 0 they output NoMsg. If the result is 1 they compute from all the messages they received some m' to take as their output. This protocol clearly has Termination. Your job is to decide how to compute the votes for the BA and the message m' when the result is 1. Argue Validity and Agreement.

Exercise 8.4 (Better Broadcast from BA, AC) Show how to broadcast a 1000-bit message using only a single BA. Assume that you do not have access to signatures, you only have authenticated channels and round-based communication. You are also allowed to use one run of BA. Make the protocol secure against $t < n/3$ Byzantine corruptions. Argue Agreement, Validity and Agreement. Try to make your protocol use as few rounds as possible.

8.7 Lower Bound on Resilience for Broadcast using Authenticated Channels*

Above we constructed a broadcast protocol for $t < n/3$ Byzantine corruptions using only authenticated channels. Since we can handle all the way to $t \le n$ when using signatures we could ask ourselves if not we could do better that $t < n/3$ from authenticated channels. Could we for instance tolerate just $t = n/3$ corrupted parties? It turns out that $t < n/3$ is the best we can do. In this section we present a proof of this. What we prove is that if $n = 3t$ then no protocol for Broadcast cannot tolerate t Byzantine corruptions.

We first prove the result for $n = 3$ and $t = 1$. Assume for the sake of contradiction that we have a protocol π with three parties $\mathsf{P}_1, \mathsf{P}_2, \mathsf{P}_3$ which solves the broadcast problem with tolerance against $t = 1$ Byzantine corruptions. Assume the protocol is for the model with pairwise authenticated channels. The protocol is illustrated as the upper left system in Fig. 8.13. We now describe a (rather strange) way in which a corrupted broadcaster might behave, let us call him $\widetilde{\mathsf{P}_1}$. What he will do is to run two copies of the *honest* algorithm for P_1: one with input 0 and one with input 1. Call them P_1^0 and P_1^1. When executing the protocol, they will be interconnected as follows:

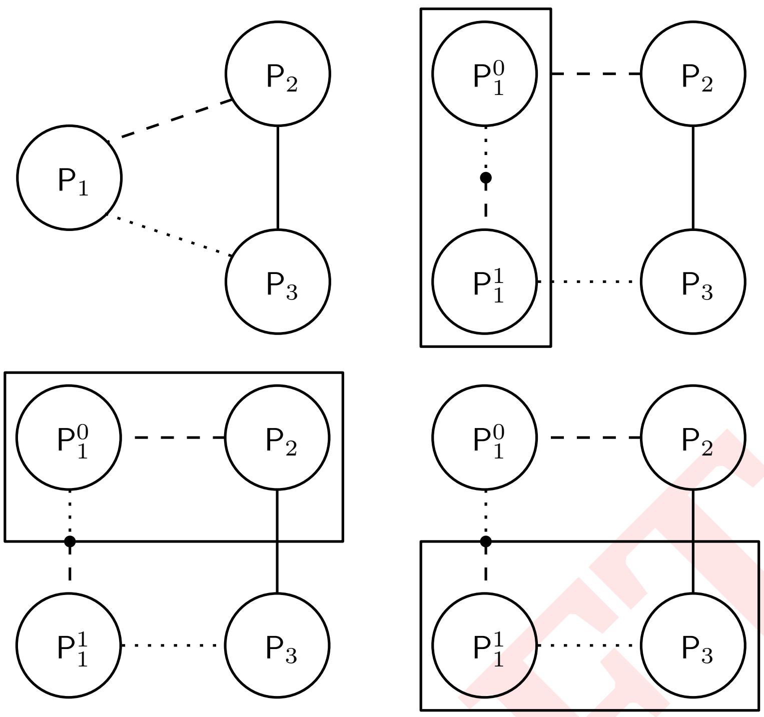


Figure 8.13 Illustration of the Proof

- Whenever P_1^0 tries to send a message to P_3, instead send it to P_1^1. Whenever P_1^0 receives a message from P_1^1 treat it as if it comes from P_3.
- Whenever P_1^1 tries to send a message to P_2, instead send it to P_1^0. Whenever P_1^1 receives a message from P_1^0 treat it as if coming from P_2.

Finally, connect $\widetilde{\mathsf{P}_1}$ to to P_2 and P_3 as follows:

- Connect P_1^0 to P_2 via the channel they normally would use.
- Connect P_1^1 to P_3 via the channel they normally would use.

The result is illustrated in the upper right corner of Fig. 8.13. Recall that even if there are actually four parties in the protocol, we may think of $\widetilde{\mathsf{P}_1} = \{\mathsf{P}_1^0, \mathsf{P}_1^1\}$ as one corrupted party. Since π can tolerate one Byzantine corruption it follows from the agreement property that whatever $\widetilde{\mathsf{P}_1}$ is doing, party P_2 and P_3 will get the same output. So when this strange four-party system is run we can conclude that $r_2 = r_3$

Consider now the system in the lower-left corner. It is the *same* system as before, but now we consider $\{\mathsf{P}_1^0, \mathsf{P}_2\}$ as a single corrupted version of P_2. Note, in particular, that it is connected to P_1^1 and P_3 exactly as P_2 would be. Since P_1^1 is running honestly as broadcaster with input $m = 1$ and since P_3 is honest and π tolerates one corrupted party (here P_2) it follows from the validity property that P_3 will output 1. Furthermore, since P_3's view of the protocol is completely the same in this case as in the former case where P_1 was corrupt, his output is the same in both cases. So we conclude that $r_3 = 1$.

Consider now the system in the lower-right corner. It is the same system as before, but now we consider $\{\mathsf{P}_1^1, \mathsf{P}_3\}$ as a single corrupted version of P_3. Since P_1^0 is running honestly as broadcaster with input $m = 0$ and since P_2 is honest and π tolerates one corrupted party (here P_3) it follows from the validity property that P_2 will output $r_2 = 0$.

From the last two ways of looking at the system, we can conclude that $0 = r_2 \neq r_3 = 1$, but we also concluded before that $r_2 = r_3$ which is of course a contradiction. So it must be that no 3-party protocol exists that tolerate 1 Byzantine corruption.

Exercise 8.5 (relaxing validity and agreement) One can make a relaxed version of broadcast, called ϵ-broadcast, where termination still must hold, but where we only require that agreement holds with probability ϵ and that validity holds with probability ϵ. Argue that it is impossible to get $\frac{3}{4}$-broadcast with $n = 3$ parties which tolerates $t = 1$ Byzantine agreement.

Exercise 8.6 (more parties) Show that it holds for all $t > 0$ and $n = 3t$ that it is impossible to get broadcast with n parties which tolerates t Byzantine corruptions in the round-based synchronous model with authenticated channels. [Hint: Assume for the sake of contradiction that such an n-party protocol exists. Group the n parties into three groups of t parties and use them to construct a 3-party protocol for broadcast tolerating 1 Byzantine corruption.]

8.8 Lower Bound on Round-Complexity of Byzantine Agreement using Signatures

In this section we prove another lower bound. Recall that the $\mathsf{DolevStrong}$ protocol runs in $n + 1$ rounds, i.e., there are $n + 1$ rounds of communication. One should wonder if one really need to use that many rounds. It turns out that if you do not want to tolerate $n-1$ Byzantine errors, but only t Byzantine errors for some lower $t < n$, then you can make a protocol running in $t + 1$ rounds. To see this, assume that a correct party receive a new SigSet in round $r = t + 2$ which is valid for m for round r. Then it contains signatures from $t + 1$ distinct parties. There are at most t corrupted parties, so SigSet contains a signature from another correct party. That party sent a set valid for m to all other correct parties in the round where it added its signature. Hence all other correct parties already received a valid set for m. Hence the correct processes have the same view already by round $t + 2$. Since they start the protocol in round 1, it takes $t + 1$ rounds to make it to round $t + 2$. Hence $t + 1$ rounds are enough to tolerate t Byzantine errors.

Amazingly it also turns out that $t + 1$ rounds are needed. At least for any protocol fulfilling these conditions:

1. The protocol only uses round-based point-to-point communication.

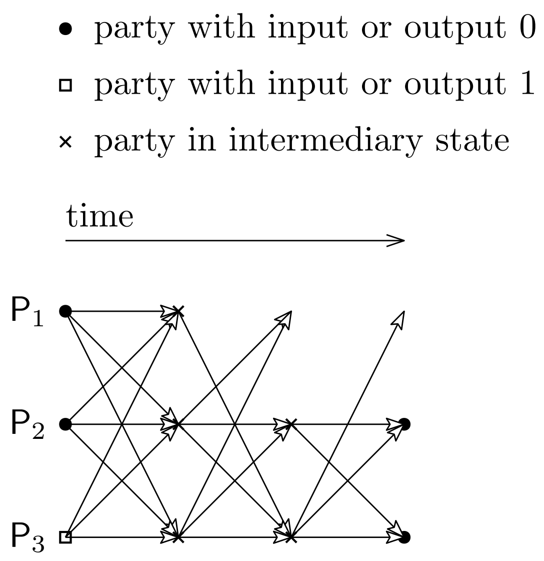

● party with input or output 0
□ party with input or output 1
× party in intermediary state
time
P_1
P_2
P_3

Figure 8.14 We use the following notation for deterministic round-based protocols where parties have binary inputs and outputs and where there might by crash-silent errors. Parties are represented by dots, boxes or crosses are and represented in columns. A dot represents a party with input or output 0. A box represents a party with input or output 1. A cross represents a party in an intermediary state. Parties in the same round are represented in a column with party P_1 at the top and P_n are the bottom. Time runs from the left to the right with each new column representing that a new round was executed. Arrows between two different parties mean that a message is sent between the two parties in that round. An arrow from one party to the same party represents that the party did not crash and thus progressed to the next round. We consider crash-silent corruptions. The lack of a dot, box or cross means the party is crashed in a previous round. We assume that when a party crashes it might send a message to some of the other parties but maybe not all. In this figure: Parties P_1 and P_2 have inputs 0 and party P_3 has input 1. In the first round no party crashes and all parties send messages to all parties. In the second round P_1 crashes and only manages to send a message to P_3. In the third round P_2 and P_3 send messages to each other. P_3 sends to P_1, but P_2 does not send to P_1 (maybe it has learned that it crashed by not receiving a message from P_1.) At the end P_2 and P_3 both output 0.

2. Ignoring the probability that the signature schemes could be broken the protocol is perfect, i.e., validity, termination and agreement hold with probability 1.

The theorem we prove is actually stronger, and it shows that the stated round complexity is necessary even in the presence of crash-silent corruptions (as you remember, we have a crash-silent failure when a process simply stops sending messages, while in a Byzantine failure the corrupted process might send arbitrary messages).

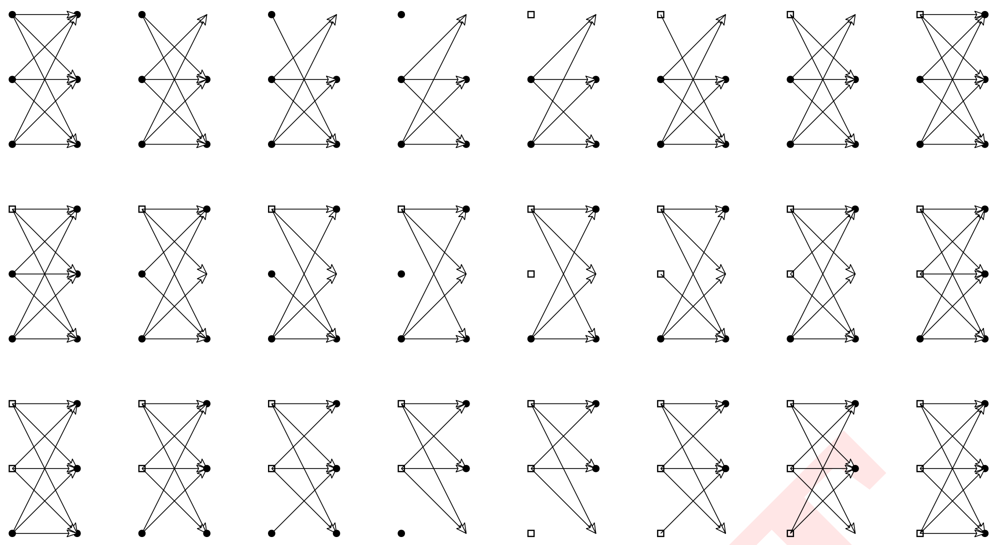


Figure 8.15 In the top-left a 1-round protocol where all parties have input 1. Assume w.l.o.g. that all parties send a message to all parties. Assume also that the protocol has agreement and validity. By validity all parties output 1 when they all have input 1, and so too in the picture. In the next picture we imagine that P_1 crashes after sending its message to P_2 and P_3. This of course cannot change that output of P_2 and P_3, so they still output 1. In the next picture it crashes before sending to P_2 but after sending to P_3. This cannot change the output of P_3 which thus still outputs 1. By agreement P_2 therefore also still outputs 1. In the next picture it crashes before sending to P_2 and P_3. This cannot change the output of P_2 as it has the same view in both runs. By agreement P_2 therefore also still outputs 1. Now P_1 is what we call isolated. Now we change the input of P_1 from 1 to 0. Since P_1 sends no messages, this cannot change the output of the other parties. Then we allow P_1 to send the message to P_3 before it crashes but not to P_2. Still P_2 outputs 1 as nothing changed for P_2 and by agreement P_3 therefore also output 1. Then we crash P_1 after sending both of its messages. P_3 still outputs 1 and by agreement P_2 also outputs 1. Then we consider the case where P_1 does not crash. Now it has an output. By agreement it must be 1. In the next row we play the exact same game, but not we isolate P_2, change its input to 0 and then bring it back to non-crashed. In the last row we play the exact same game with P_3 and thus change its input to 0. All the way the output of all parties remain 1 by agreement. But in the bottom-right we then have a run where all parties have input 0 but where they output 1. This violates validity.

Consider a deterministic, secure protocol for Cast *for n parties, where all parties always terminate no later than round r. Assume it uses* Round-Comm *for communication. Assume that up to t parties might be crash-silent corrupted and that $n \geq t+2$. Then it holds that $r \geq t+1$.*

We sketch the main intuition of the proof here, with many details omitted. The proof is by contradiction, therefore we start by assume that we have a protocol

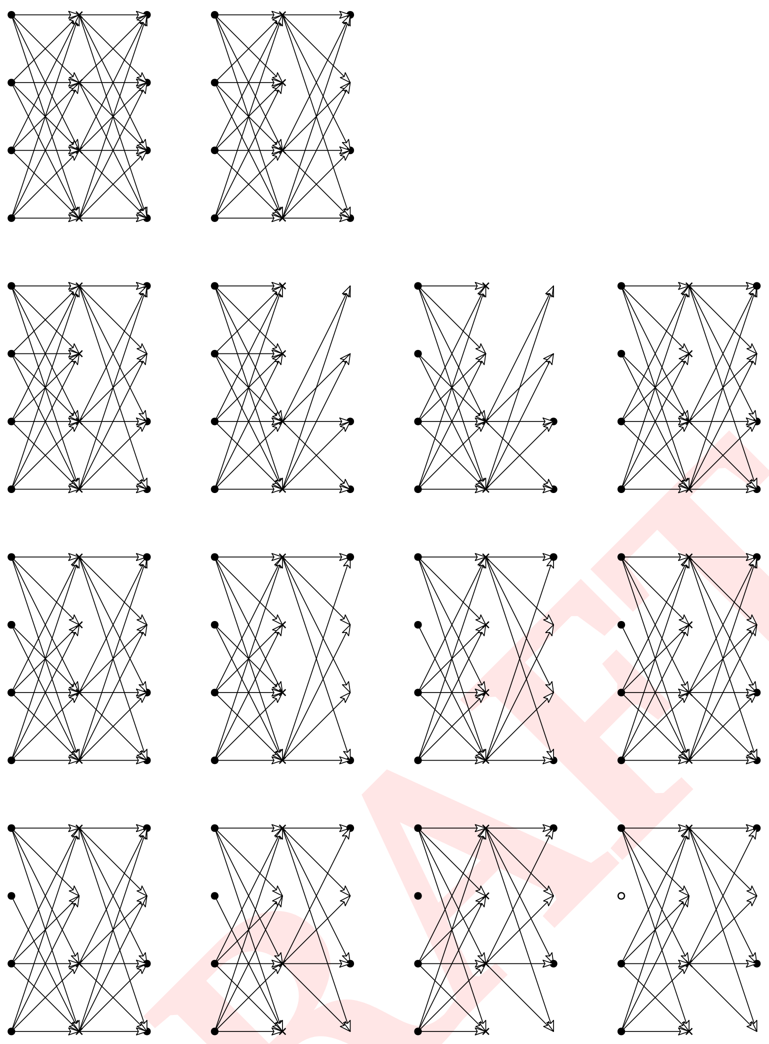


Figure 8.16 Top-left: a 2-round protocol where all parties have input 1. We show how to isolate P_2 in the first round and change its input without changing the output of the other parties. We already know how to isolate parties in round 2 using the tricks as in Fig. 8.15: we can take a party and remove all communication sent in round 2 without that affecting the output of the other parties by removing one message at a time. To save on the number of figures, we will do it in one go, as in the first row, where we remove the messages sent by P_2. In the second row we then start from that situation and in the next step remove all communication sent by P_1 in round 2. Now P_1 is isolated in round 2, so we can remove the message from P_2 to P_1 without that affecting the output of any party (as P_1 do not send any messages that could cause such an affect). Notice that to "remove" the message from P_2 to P_1 we need to crash P_2 in the first round. Therefore it was important that P_2 was already crashed in round 2, or the crash in round 1 would cause a change in the behavior of P_2 in the next round, which could affect the output of the other parties. After dropping the message from P_2 to P_1, we then bring P_1 back to life; recall that we can do so without affecting the output of the other partes. In the third row we then crash P_3 in round 2, then we drop the message from P_2 to the now isolated P_3 in round 1, and then we bring P_3 back to life. In the last row we then crash P_4 in round 2, then we drop the message from P_2 to the now isolated P_4 in round 1. Now P_2 is isolated in the first round, so now we can change its input.

which always terminates within t round and we assume that it has validity and agreement. From this we then conclude that it does not have validity, a clear contradiction. If all parties terminate before round t we can without loss of generality assume that all terminate precisely in round t. Early parties can just wait with giving their outputs. We can also assume w.l.o.g. that all parties send a message to all parties. Consider a run of the protocol where all parties have input 1. Then by validity they all output 1. We will then go through a series of thought experiments where we remove messages from the execution but still look at a run which could result from at most t crash-silent errors. In each step there is one party whose view did not change, so that party will keep outputting 1. By agreement all other parties therefore also output 1. Eventually, after we remove enough messages, we will reach a thought experiment in which one of the parties is isolated: it does not send any messages at all. When this happens we change its input to 0. Since it sends no messages, this cannot affect the output of other parties, so they keep outputting 1. In the end we have changed all inputs to 0, but still the output of all parties is 1. This violates validity.

The full proof of this is complicated, and we will not cover it in this book. We will however look at two special cases that should allow the interested reader to reconstruct the full proof. We first look at the case $t = 1$. We look at the minimal case with $n = 3$ parties. The strategy is explained in Fig. 8.15. It shows how to isolate parties, change their input and bring them back to life. The proof generalises to any n. For the case $t = 2$, see Fig. 8.16. The strategy to isolate parties P_i in round 1 is to use that we know how to isolate parties P_j in the round 2 using the strategy form Fig. 8.15. First we crash P_i in round 2 (this cost us one corruption). Then we party-by-party crash P_j in round 2 (this cost one more corruption) and then drop the message from P_i to P_j. Then we bring P_j back to life and crash the next P_j. When P_i is isolated in round 1 we change its input. We do this for all P_i and then reach the sought contradiction. For general t and a protocol running in r rounds we can change the input of P_i as follows. First we use $t-1$ corruptions to crash P_i in round 2. This gives a situation where P_i is crashed in round 1 (using thus just one corruption). This means we still have $t-1$ corruptions to work with. We can use them to crash another P_j in round 2. Then we drop the message from P_i to P_j in round 1. Then we bring P_j back to life. We do so for all P_j. Now P_i is isolated and we can change its input.

Exercise 8.7 (weak validity) There exist a weaker notion of validity called weak validity which says that validity only needs to hold if all processes are correct, i.e., if there is a single corrupted party, the output need not be the input of some honest party. Agreement is still required to hold also when there are corruptions. Argue that Theorem 8.7 also holds for protocols with agreement and weak validity.

Exercise 8.8 (Round Lower Bound for BA) Consider a deterministic, secure protocol for BA for n parties, where all parties always terminate no later than round r. Assume it uses Round-Comm for communication. Assume that up to t parties might be crash-silent corrupted and that $n \geq t + 2$. Prove that $r \geq t$.

9

Asynchronous Agreement (DRAFT)

Contents

Attack Model *In this chapter we assume that*

1. *There are n parties.*
2. *Up to t parties might be Byzantine corrupted. We will consider several limits on t, but mostly $t < n/3$.*
3. *The adversary cannot drop messages.*
4. *The adversary can delay messages arbitrarily except that it must eventually deliver all messages.*
5. *The adversary cannot inject messages. In some places we assume this by assuming authenticated channels. 'In other places it is ensured by using signatures schemes.*

9.1 Introduction

In this chapter we study protocols which allow several parties in a distributed system to agree on a decision even in the presence of a Byzantine adversary which tries to make them disagree. As in Chapter 8 we will study broadcast and Byzantine Agreement. This difference in this chapter is that we will study the problems in the asynchronous model. This main characteristic of this model is that messages can be delayed for arbitrarily long. This means that if a party is supposed to send you a message and it does not arrive, you cannot conclude that the party is corrupted and did not send the message. It might also just be that the message is very slow. This makes it harder to make secure protocols.

Recall that when we solve agreement problems we are interested in three properties.

Agreement All honest parties make the same decision.

Validity The decision that is made must be sensible in some sense.

Termination If all parties starts running the protocol then all honest parties eventually end up making a decision.

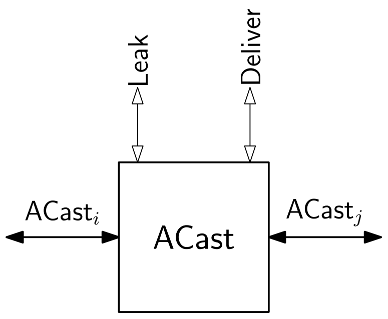
Below we will define these properties more carefully for asynchronous broadcast and asynchronous Byzantine agreement. We will give examples of secure protocols for both tasks. Finally we will prove that it is impossible to solve Asynchronous Byzantine Agreement without using random choices.

9.2 Defining Asynchronous Broadcast

The ideal functionality for asynchronous broadcast in given in Fig. 9.1. It is very similar to the synchronous one, so we will not discuss it much. A main difference is that we only require the termination to hold when all parties started running the protocol. But we do not consider it part of the user contract that all honest parties start running the protocol. Stating that all honest parties start running the protocol as pre condition for a safety properties is not reasonable in the asynchronous as some parties might fall arbitrarily long behind. Instead we just say that the protocol need not guarantee any liveness properties until all honest parties started running it.

9.3 Asynchronous Broadcast from Signatures

We first present a simple broadcast protocol using signatures. It uses the simple PKI toyPKI for simplicity. In reality it would need access to a real PKI. It works for n parties out of which at most $t < n/3$ are corrupted. Note that this implies that $n \geq 3t + 1$ and $n - t \geq 2t + 1$. Basically the broadcaster, P_i, will ask all n parties to sign the message m that it wants to send. All parties will grant such a signature and will sign at most one m for P_i. Then P_i waits for $n - t$ parties to send a signature. These signatures will eventually arrive as there are at least $n-t$ honest parties and they all sign m for P_i. To broadcast m, the sender, P_i, will send m along with $n-t$ signatures on m. A receiver will only output m if it is received along with $n - t$ signatures from distinct parties. If any party outputs m, then it passes on m to all other parties along with the signatures to make them also output m. We sketch the arguments for Agreement, Validity and Termination.



Ports: For each part P_i there is a protocol port ACast_i. There are also special ports leak and $\mathsf{Deliver}$.

Syntax: The inputs on ACast_i from parties P_i can be of the form $(\mathsf{bid}, \mathsf{P}_i, m)$, where bid is a broadcast ID and m is a message. Furthermore, parties $\mathsf{P}_j \neq \mathsf{P}_i$ can give inputs of the form $(\mathsf{bid}, \mathsf{P}_i, ?)$ indicating that they know that P_i is about to give an input. Parties P_j can give outputs of the form $(\mathsf{bid}, \mathsf{P}_i, m)$ indicating that they think P_i has broadcast m.

Input: On input $(\mathsf{bid}, \mathsf{P}_i, m)$ on ACast_i the IF outputs $(\mathsf{bid}, \mathsf{P}_i, m)$ on leak.

Output: On input $((\mathsf{bid}, \mathsf{P}_i, m), \mathsf{P}_j)$ on $\mathsf{Deliver}$ the IF outputs $(\mathsf{bid}, \mathsf{P}_i, m)$ on ACast_j.

User Contract: If an honest party is honest then for each $(\mathsf{bid}, \mathsf{P}_j)$ it gives at most one input of the form $(\mathsf{bid}, \mathsf{P}_j, \cdot)$.

Ideal Functionality Contract: If the inputs are according to the user contract, then the adversary must make the IF give outputs according to the following rules.

IO: Input before Output: If P_j is honest and some $(\mathsf{bid}, \mathsf{P}_i, \cdot)$ was output on ACast_j, then previously some $(\mathsf{bid}, \mathsf{P}_i, \cdot)$ was input on ACast_j.

Validity: If P_i and P_j are honest and $(\mathsf{bid}, \mathsf{P}_i, m)$ was output on ACast_j, then previously some $(\mathsf{bid}, \mathsf{P}_i, m)$ was input on ACast_i.

Agreement: If P_j and P_k are honest and $(\mathsf{bid}, \mathsf{P}_i, m_j)$ was output on ACast_j and $(\mathsf{bid}, \mathsf{P}_i, m_k)$ was output on ACast_k, then $m_j = m_k$.

Termination: If all honest parties give an input of the form $(\mathsf{bid}, \mathsf{P}_i, \cdot)$, then the following holds:

1. If P_i and P_j are honest and an input of the form $(\mathsf{bid}, \mathsf{P}_i, \cdot)$ is input on ACast_i, then eventually an output of the form $(\mathsf{bid}, \mathsf{P}_i, \cdot)$ is output on ACast_j.
2. If P_j and P_k are honest and an output of the form $(\mathsf{bid}, \mathsf{P}_i, \cdot)$ is given on ACast_j, then eventually an output of the form $(\mathsf{bid}, \mathsf{P}_i, \cdot)$ is given on ACast_k.

Figure 9.1 An ideal functionality ACast for Asynchronous Broadcast.

Agreement

Assume that two honest parties P_j and P_k output m_j and m_k. We want to show that $m_j = m_k$ even if there are t corrupted parties maybe even including the sender P_i. Observe first that each of the $n - t$ honest parties signs at most $n - t$ messages. This gives at most $n - t$ distinct signatures. The t corrupted parties

All parties P_i run the following activation rules:

Get Keys: The protocol uses the network resource toyPKI. The first time a party is activated it queries toyPKI and learns the public keys $(\mathsf{vk}_1, \ldots, \mathsf{vk}_n)$ of all parties and its own secret key sk_i. Then let $\mathsf{SignedBy}_{\mathsf{bid}}$ be the empty set for all broadcast IDs bid.

Request Signatures: On input $(\mathsf{bid}, \mathsf{P}_i, m)$ on ACast_i compute $\sigma_{\mathsf{bid},i} = \mathsf{Sig}_{\mathsf{sk}_i}(\mathsf{bid}, \mathsf{P}_i, m)$ and send $(\sigma_i, (\mathsf{bid}, \mathsf{P}_i, m))$ to all parties. We call a value $(\sigma_i, (\mathsf{bid}, \mathsf{P}_i, m))$ a valid request from P_i if $\mathsf{Ver}_{\mathsf{vk}_i}(\sigma_i, (\mathsf{bid}, \mathsf{P}_i, m)) = \top$.

Grant Signatures: On input $(\mathsf{bid}, \mathsf{P}_j, ?)$ on ACast_i and a valid request $(\sigma_j, (\mathsf{bid}, \mathsf{P}_j, m))$ from P_j, where no other valid request was received for a value of the form $(\mathsf{bid}, \mathsf{P}_j, \cdot)$, compute $\sigma_{\mathsf{bid},i} = \mathsf{Sig}_{\mathsf{sk}_i}(\mathsf{bid}, \mathsf{P}_j, m)$, and send $\sigma_{\mathsf{bid},i}$ to P_j. We call $\sigma_{\mathsf{bid},i}$ a grant for $(\mathsf{bid}, \mathsf{P}_j, m)$ from P_i if $\mathsf{Ver}_{\mathsf{vk}_i}(\sigma_{\mathsf{bid},i}, (\mathsf{bid}, \mathsf{P}_j, m)) = \top$.

Collect Signatures: On input $(\mathsf{bid}, \mathsf{P}_j, ?)$ on ACast_i and a grant $\sigma_{\mathsf{bid},j}$ for $(\mathsf{bid}, \mathsf{P}_i, m)$ from $\mathsf{P}_j \notin \mathsf{SignedBy}_{\mathsf{bid}}$ add P_j to $\mathsf{SignedBy}_{\mathsf{bid}}$ and store $\sigma_{\mathsf{bid},j}$.

Send Signatures: On input $(\mathsf{bid}, \mathsf{P}_i, m)$ on ACast_i and when $|\mathsf{SignedBy}_{\mathsf{bid}}| = n - t$ send $(\mathsf{bid}, \mathsf{P}_i, m)$ along with $\mathsf{SigSet}_i = \{\sigma_{\mathsf{bid},j}\}_{\mathsf{P}_j \in \mathsf{SignedBy}_{\mathsf{bid}}}$ to all parties. We call SigSet_i a valid signature set for $(\mathsf{bid}, \mathsf{P}_i, m)$ if it contains $n-t$ signatures $\sigma_{\mathsf{bid},j}$ from distinct parties which are all valid for $(\mathsf{bid}, \mathsf{P}_i, m)$.

Output: On $(\mathsf{bid}, \mathsf{P}_j, m)$ and a valid signature set SigSet_j for $(\mathsf{bid}, \mathsf{P}_j, m)$ from any party, where no output of the form $(\mathsf{bid}, \mathsf{P}_j, \cdot)$ was given before, output $(\mathsf{bid}, \mathsf{P}_j, m)$ on ACast_i and send $(\mathsf{bid}, \mathsf{P}_j, m)$ and SigSet_j to all parties.

Figure 9.2 PKI2Cast

might sign both m_j and m_k, which might give at most $2t$ more distinct signatures. This gives a total of at most

$$(n-t) + 2t = n + t$$

distinct signatures on m_j or m_k. Recall then that P_j saw $n-t$ distinct signatures on m_j and P_k saw $n-t$ distinct signatures on m_k. Otherwise they would not have output the messages. If $m_j \neq m_k$ then all signatures on m_j are different form the ones on m_k. So in that case there are $(n-t)+(n-t)$ distinct signatures. We have that

$$(n-t) + (n-t) = (n-t) + (2t+1) = n + t + 1 \ .$$

So to have $m_j \neq m_k$ we need P_j and P_k to have a total of at least $n+t+1$ distinct signatures. But we concluded above that P_j and P_k have a total of at least $n+t$ distinct signatures. So we conclude that it is not the case that $m_j \neq m_k$, which gives us that $m_j = m_k$, as desired.

Validity

This is simple: honest parties only output m as coming from the broadcaster P_i if it sees $n-t$ signatures on m. But then it saw at least $(n-t)-t = t+1$ signatures from honest parties. But then $t+1 \geq 1$ honest parties signed m. And honest parties only sign the m that comes from P_i.

IO

This follows as P_j by construction only give an output for $(\mathsf{bid}, \mathsf{P}_i, \cdot)$ if it got an input of that form.

Termination 1

If P_i is honest it asks all parties to sign m. At least the $n-t$ honest will grant the signatures and hence P_i will at some point receive $n-t$ signatures on m. Then P_i will forward them, so eventually all honest P_j receive m along with $n-t$ signatures, at which point they output m.

Termination 2

When a party outputs m it forwards m and the $n-t$ distinct signatures, which makes all other honest parties output m too.

9.4 Asynchronous Broadcast from Authenticated Channels

In this section we look at a Byzantine broadcast protocol, called (Bracha Broadcast), for the fully connected model with authenticated channels and asynchronous communication with eventual delivery. Recall that in the asynchronous model we do not use time and there is no known upper bound on the delivery time of messages. We do not assume signatures, so we cannot just use the above protocol.

The protocol was first discovered by Gabriel Bracha in 1984[2] and is arguably one of the most elegant asynchronous protocols. It concisely illustrates some concepts and techniques crucial for programming the asynchronous model, so we present it here and elaborate on some of these points.

Below we will simplify notation a bit. We will assume that it is always P_1 who is the broadcaster. When P_1 gets input $(\mathsf{P}_1, \mathsf{bid}, m)$ on ACast_1 to start a broadcast, we say that it got input $(\textsc{Broadcast}, \mathsf{P}_1, \mathsf{bid}, m)$. We assume that no other parties acts before having gotten input $(\mathsf{P}_1, \mathsf{bid}, ?)$. When a party outputs $(\mathsf{P}_1, \mathsf{bid}, m)$ on ACast_j we say that it has output $(\textsc{Deliver}, \mathsf{P}_1, \mathsf{bid}, m)$. This way we do not have to keep naming the ports.

Let n be the number of parties and let $t < n/3$ be the number of Byzantine corruptions to tolerate. The protocol is presented in Fig. 9.4. Note that if you receive $(\textsc{Send}, \mathsf{P}_1, \mathsf{bid}, m)$ and $(\textsc{Send}, \mathsf{P}_1, \mathsf{bid}, m')$ for $m' \neq m$, then you only echo the first one. This ensures that the broadcaster only is able to send one m per bid.

Before we analyse the protocol, we want to highlight some general principles that are important for any fully asynchronous protocol: there are no rounds, only activation rules, there is a point where parties wait for message from $n-t$ parties, there is a point where they wait for messages from $t+1$ parties, and we have that $n > 3t$ to get a big overlap between the information of honest parties. Let us take them one by one.

Send P_1: On input $(\textsc{Broadcast}, \mathsf{P}_1, \mathsf{bid}, m)$, send $(\textsc{Send}, \mathsf{P}_1, \mathsf{bid}, m)$ to all parties.
Echo P_i: On message $(\textsc{Send}, \mathsf{P}_1, \mathsf{bid}, m)$ from P_1, send $(\textsc{Echo}, \mathsf{P}_1, \mathsf{bid}, m)$ to all parties. This is not done if you earlier received a message of the form $(\textsc{Send}, \mathsf{P}_1, \mathsf{bid}, \cdot)$
Ready 1 P_i: Once message $(\textsc{Echo}, \mathsf{P}_1, \mathsf{bid}, m)$ has been received from $n-t$ parties, send $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m)$ to all parties if not already done.
Ready 2 P_i: Once message $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m)$ has been received from $t+1$ parties, send $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m)$ to all parties if not already done.
Deliver P_i: Once message $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m)$ has been received from $n-t$ parties, output $(\textsc{Deliver}, \mathsf{P}_1, \mathsf{bid}, m)$ and terminate the protocol.

Figure 9.3 Bracha Broadcast

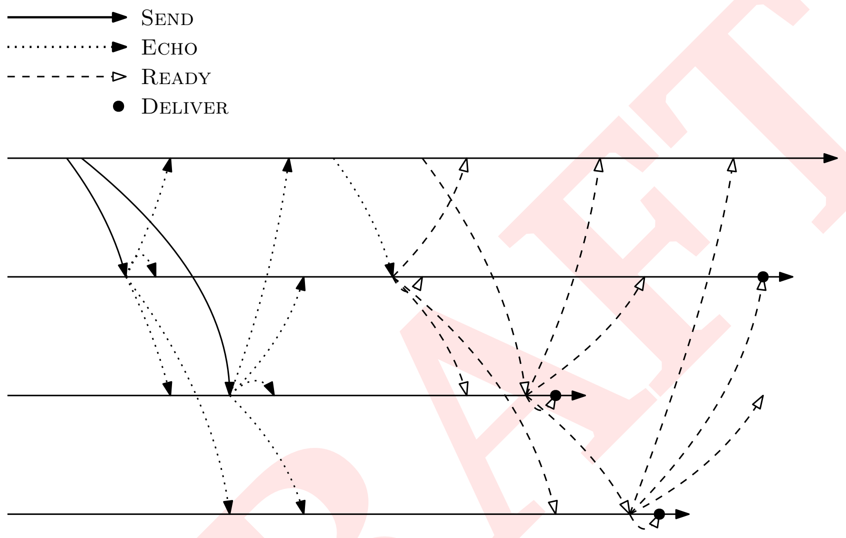


Figure 9.4 Example execution of Bracha Broadcast with a corrupted P_1.

No Rounds

In an asynchronous protocol there are no rounds. Each of the rules Send, *Echo*, *Ready 1*, *Ready 2*, and *Deliver* are independent activation rules. They trigger when their precondition is fulfilled. Recall furthermore the convention that if several activation rules have their precondition fulfilled, then the topmost one will be activated.

The distinction between round and activation rules is important. Note in particular that it might easily happen that a party triggers *Ready 2* before it triggers *Ready 1*. It might even happen that a party triggers *Deliver* and terminates before triggering *Ready 1*; In that case it will never trigger *Ready 1*. It might seem dangerous that some party terminates before running *Ready 1*, but in fact the protocol works exactly because we allow this. Consider a run with four parties.

Assume that P_1 broadcasts and is corrupted. Note that $n = 4$ and $t = 1$, so $n - t = 3$. Consider the execution in Fig. 9.4:

1. P_1 sends $(\textsc{Send}, \mathsf{P}_1, \mathsf{bid}, m)$ to P_2 and P_3 only.
2. $\mathsf{P}_2, \mathsf{P}_3$ send $(\textsc{Echo}, \mathsf{P}_1, \mathsf{bid}, m)$ to all parties.
3. $\mathsf{P}_2, \mathsf{P}_3, \mathsf{P}_4$ all receive the two $(\textsc{Echo}, \mathsf{P}_1, \mathsf{bid}, m)$-messages.
4. P_1 sends $(\textsc{Echo}, \mathsf{P}_1, \mathsf{bid}, m)$ to P_2 only.
5. P_2 receives the third $(\textsc{Echo}, \mathsf{P}_1, \mathsf{bid}, m)$-message.
6. P_2 triggers *Ready 1* and sends $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m)$ to all parties.
7. $\mathsf{P}_2, \mathsf{P}_3, \mathsf{P}_4$ all receive the $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m)$-message from P_2.
8. P_1 sends $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m)$ to P_3.
9. P_3 receives the second $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m)$-message and triggers *Ready 2* and sends $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m)$ to all parties.
10. P_3 receives the third $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m)$-message and triggers *Deliver*.
11. P_4 receives the second $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m)$-message and triggers *Ready 1* and sends $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m)$ to all parties.
12. P_4 receives the third $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m)$-message and triggers *Deliver*.
13. P_2 receives the third $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m)$-message and triggers *Deliver*.

Gathering Maximal Information

If you wait for message from $n - t$ processes you are trying to collect as much information as you can without deadlocking. There are n parties in total and t of them might be corrupt and never send their messages. So if you wait for $n - t + 1$ messages, then the last one might never arrive. And remember that in the asynchronous model, there is no way to distinguish a message that was not sent from a message that was sent by an honest process but is just very slow. So you can never wait for more than $n - t$ message. In the above protocol the parties wait for the same message being echoed from $n - t$ parties to ensure that they all received the same m, as they should. We would of course have liked to check that all n did so, but we cannot wait for n messages. This is illustrated in Fig. 9.5.

No Timeout, Better Security

As a consequence of not having timeouts, protocols in the asynchronous model are in general more secure (all other things being equal). To illustrate this point, consider Fig. 9.6. Notice that both P_1 and P_2 have a slow messages. If we had been in the synchronous model and the picture was after the timeout of the round had passed, then both P_1 and P_2 would be considered corrupted. However, in the asynchronous model there are no timeouts, so P_1 and P_2 are still considered correct. They just have some slow messages, which is allowed. It is therefore easier for a party to be considered correct in the asynchronous model. So if we have two protocols, one for the synchronous model and one for the asynchronous model and they are both secure against for instance $t < n/3$ corruptions, then the one for the asynchronous model is more secure, as it is easier to be considered correct in the asynchronous model.

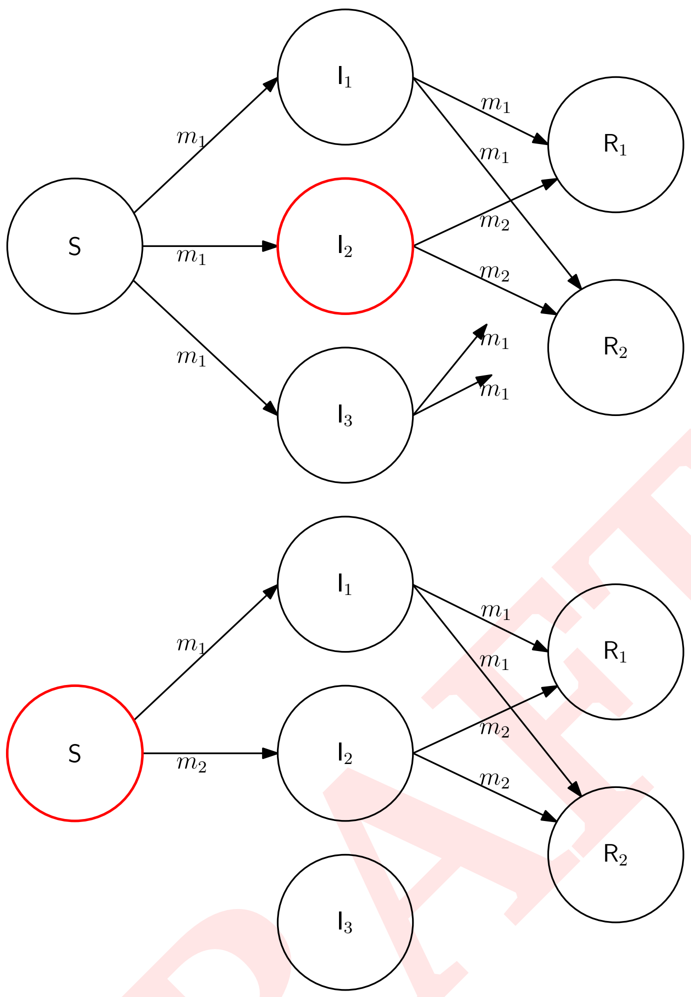


Figure 9.5 The transferability problem gets harder in asynchronous networks as you cannot wait for all messages. To see this consider the protocol from Fig. 8.9 solving transferability using three intermediaries. In the top row the second intermediary is corrupted and changes its message. At the same time the messages from the third intermediary are slow. If only the receives could wait for those messages, then they could take majority. But consider the second row. There the sender is corrupted and sent different messages to I_1 and I_2 and never sent anything to I_3. In lack of timeouts I_3 can never react to that, so nothing will ever be sent by I_3. So if the receivers would wait for a message from I_3, then they would wait forever.

Hearing from a Correct Party

If you wait for messages from $t+1$ parties, then you are essentially only ensuring that you hear from at least one correct party. There are at most t corrupt parties, so if you get $t+1$ messages from distinct parties, at least one of them was sent by a correct one. In the above protocol this is used to learn that some honest party

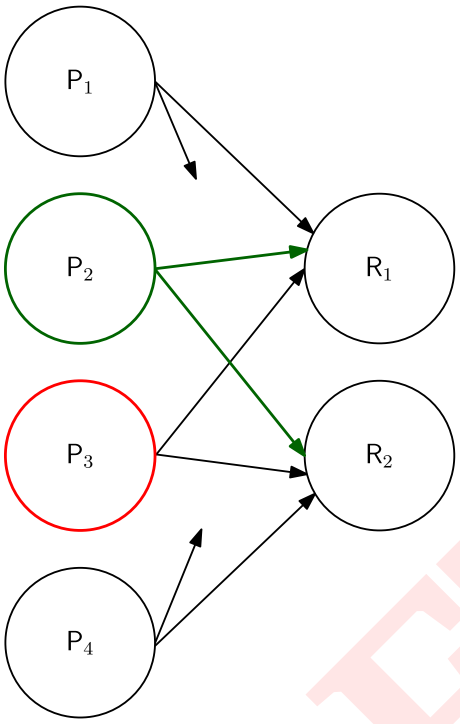


Figure 9.6 With four parties and at most one corrupted party, receivers can wait for three messages and be guaranteed to have heard from at least one common honest party, here P_2. One cannot do with less than four, as different messages can be slow towards different receivers, so they cannot be guaranteed to hear from the same three parties. And they might be unlucky that the corrupted party is among the two common parties they hear from.

sent $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m)$, from which we conclude that some correct process saw the same m from $n - t$ other parties. We will see below how that is used.

Hearing from a Common Correct Party

Why the $n > 3t$ condition? Well, if two different parties P_1 and P_2 each hear from $n-t$ parties, then the number of parties they both heard from might be as low as $n-2t$: it could for instance be that P_1 heard from $\mathsf{P}_1, \dots, \mathsf{P}_{n-t}$ and that P_2 heard from $\mathsf{P}_{t+1}, \dots, \mathsf{P}_n$, giving the shared "informants" $\mathsf{P}_{t+1}, \dots, \mathsf{P}_{n-t}$, and if $n = 3t$, then $n-t-(t+1)+1 = n-2t$. Out of these $n-2t$ there might be t parties that are corrupted. So, the number of honest parties they both heard from might be as low as $n-3t$. So, we need $n > 3t$ to guarantee that there is at least one shared correct party they both heard from. This is illustrated in in Fig. 9.6.

It turns out that if this was not the case, it might be that information never was able to pass between P_1 and P_2, and then they cannot accomplish anything. To see this, assume that $n = 3t$, and divide the network into three groups of parties, A, B and C, each of size t. Assume that parties in B are corrupted. Remember that honest parties have to proceed after hearing from $n - t$ parties and that $n - t = 2t$. This means that if the adversary delays messages from C long enough and meanwhile lets the parties in B talk to parties in A without delay, then the protocol will eventually terminate. So, the parties in A are willing to terminate the protocol without hearing from the parties in C. Similarly, the parties in C are willing to terminate the protocol without ever hearing from the parties in A. So in conclusion, if the adversary delays messages between A and C for long enough and meanwhile lets the corrupt parties in B talk to both A and C as described, all the honest parties in A and C will terminate, but no information was ever exchanged between the groups A and C. So clearly they cannot accomplish any interesting coordination, let alone achieve agreement.

Running Forever?

A particular thing to notice about Bracha Broadcast is that if for instance P_1 is corrupt and does not send its message, then the correct parties run forever. The reason is that they will forever be waiting in *Echo*. And there is no way they can avoid this. They cannot distinguish a message not sent by a corrupt party from a message sent by a correct party that was delayed arbitrarily. So if they terminate before getting something from P_1, they will jeopardise the safety of the protocol when P_1 is honest but "slow".

9.4.1 Analysing Bracha Broadcast

We now proceed to analyse the protocol.

Theorem 9.1 *Bracha Broadcast securely implements* $\mathsf{ACast}n$ *when there are* $t < n/3$ *Byzantine corruptions.*

PROOF We argue Termination 1. We have to argue that of P_1 is honest and broadcast, then all parties will eventually make an output. If P_1 gets input $(\textsc{Broadcast}, \mathsf{P}_1, \mathsf{bid}, m)$ it sends $(\textsc{Send}, \mathsf{P}_1, \mathsf{bid}, m)$ to all parties. The $n-t$ honest parties will eventually get the message and send $(\textsc{Echo}, \mathsf{P}_1, \mathsf{bid}, m)$ to all parties. Therefore all $n - t$ honest parties will get $n - t$ such messages and send $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m)$ to all parties. Therefore all honest parties eventually get $n-t$ such messages and output $(\textsc{Deliver}, \mathsf{P}_1, \mathsf{bid}, m)$.

IO is by construction: we assumed that honest party started acting before it got an input $(\mathsf{bid}, \mathsf{P}_1, \cdot)$.

We argue Validity. We have to argue that if P_1 is honest, then no honest party outputs $(\textsc{Deliver}, \mathsf{P}_1, \mathsf{bid}, m)$ unless P_1 sent m. If P_1 never got input $(\textsc{Broadcast}, \mathsf{P}_1, \mathsf{bid}, m)$, then clearly no correct P_i ever sent $(\textsc{Echo}, \mathsf{P}_1, \mathsf{bid}, m)$. Hence at most t parties sent $(\textsc{Echo}, \mathsf{P}_1, \mathsf{bid}, m)$. Since $n - t > t$ it follows that no correct P_i ever sent $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m)$. Therefore no correct P_i outputs $(\textsc{Deliver}, \mathsf{P}_1, \mathsf{bid}, m)$.

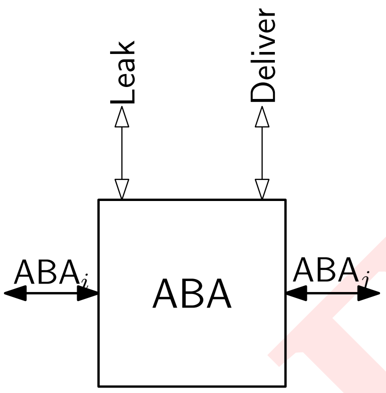
We argue Termination 2. We have to argue that if some honest party terminates then all honest parties eventually terminate, even if P_1 is corrupt. The above cases were fairly simple. They did not use all the details of the protocol. This will happen now. The reason this is harder than it looks is that P_1 might be corrupted. So we start from the assumption that a correct P_i outputs $(\textsc{Deliver}, \mathsf{P}_1, \mathsf{bid}, m')$ (we now write m' since, as P_1 is corrupted, we do not know what the input of P_1 was). Then we have to analyse why this could happen. Then we need to argue that in those cases it would force any other correct P_j to also output $(\textsc{Deliver}, \mathsf{P}_1, \mathsf{bid}, m')$. So, assume that some P_i outputs $(\textsc{Deliver}, \mathsf{P}_1, \mathsf{bid}, m')$. Inspecting the code we see that then P_i must have received $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m')$ from $n - t$ parties. We have that $n > 3t$, so $n - t > 2t$. So P_i must have received $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m')$ from at least $2t + 1$ parties. There are at most t corrupted parties, so we can conclude that P_i must have received $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m')$ from at least $t + 1$ correct parties. By inspection of the code we see that when these $t + 1$ parties sent $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m')$ to P_i, they also sent $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m')$ to all other parties. Therefore all correct parties will eventually receive $t + 1$ messages of the form $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m')$. Hence they will all trigger Ready 2 and send $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m')$ to all other parties. Now *all* of the $n - t$ honest parties sent $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m')$ to all honest parties. Therefore all honest parties will eventually receive $n - t$ messages of the form $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m')$ and output $(\textsc{Deliver}, \mathsf{P}_1, \mathsf{bid}, m')$. What a beautiful protocol!

We then argue Agreement. Assume that there are exactly t corrupted parties and that $n = 3t + 1$. It only makes the argument harder to assume that the number of corruptions is the maximal allowed. Note that there are then $2t + 1$ honest parties. Assume that P_i outputs $(\textsc{Deliver}, \mathsf{P}_1, \mathsf{bid}, m_i)$ Inspecting the code we see that then P_i must have received $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m_i)$ from $n - t$ parties. We have that $n > 3t$, so $n - t > 2t$. So P_i must have received $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m_i)$ from at least $2t + 1$ parties. There are at most t corrupted parties, so we can conclude that P_i must have received $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m_i)$ from at least $t + 1$ correct parties. Similarly, if P_j outputs $(\textsc{Deliver}, \mathsf{P}_1, \mathsf{bid}, m_j)$, then P_j received $(\textsc{Ready}, \mathsf{P}_1, \mathsf{bid}, m_j)$ from at least $t + 1$ correct parties. Since $(t + 1) + (t + 1) = 2t + 2 > 2t + 1$ it follows that there is at least one honest parties who sent $(\textsc{Deliver}, \mathsf{P}_1, \mathsf{bid}, m_i)$ to P_i and $(\textsc{Deliver}, \mathsf{P}_1, \mathsf{bid}, m_j)$ to P_j. Since honest parties forward only one message of the form $(\textsc{Deliver}, \mathsf{P}_1, \mathsf{bid}, \cdot)$ per bid it follows that $m_i = m_j$, as we had to prove. □

The above protocol tolerates $t < n/3$ Byzantine corruptions. This can be seen to be optimal as it is optimal already in synchronous networks. However, it turns out that the bound holds in a much stronger sense in asynchronous networks. The following exercise asks you to explore this.

Exercise 9.1 (Optimality of the Corruption Bound) Assume a fully connected model with authenticated channels and asynchronous communication with eventual delivery. Assume that parties are allowed to use signatures too. Assume that there are Byzantine errors. Show that there is no protocol for broadcast in this model tolerating $t = n/3$ errors. Start by looking at the case $n = 3$ parties and $t = 1$ Byzantine error. The main idea to the proof is that it is impossible to distinguish an arbitrarily slow correct party and a corrupted party. So if the sender is corrupted and collaborates with $t-1$ other corrupted parties and slows down t of the correct parties completely, then the other t parties must be willing to run and terminate the protocol with the t corrupted parties. Why is this? And how do you exploit it to break agreement?

△



Ports: For each part P_i there is a protocol port ABA$_i$. There are also special ports leak and Deliver.

Input: On input $(\textsc{Vote}, \mathsf{baid}, v)$ on ABA$_i$ where $v \in \{0,1\}$ the IF outputs $(\textsc{Vote}, \mathsf{baid}, v)$ on leak.

Output: On input $(\textsc{Decision}, \mathsf{baid}, d), \mathsf{P}_j)$ on Deliver where $d \in \{0,1\}$ the IF outputs $(\textsc{Decision}, \mathsf{baid}, d)$ on ABA$_j$.

User Contract: If a party is honest then for each baid it gives at most one input of the form $(\textsc{Vote}, \mathsf{baid}, \cdot)$.

Ideal Functionality Contract: If the inputs are according to the user contract, then the IF gives outputs according to the following rules. It does so by rejecting outputs from the adversary breaking these properties.

- **IO:** If P_j is honest and some $(\textsc{Decision}, \mathsf{baid}, \cdot)$ was output on ABA$_j$, then previously some $(\textsc{Vote}, \mathsf{baid}, \cdot)$ was input on ABA$_j$.
- **Validity:** If for some baid some correct P_i gives an output $(\textsc{Decision}, \mathsf{baid}, d)$, then at some earlier point some correct P_j had the input $(\textsc{Vote}, \mathsf{baid}, d)$.
- **Agreement:** If for some baid some correct P_i and P_j output $(\textsc{Decision}, \mathsf{baid}, d_i)$ and $(\textsc{Decision}, \mathsf{baid}, d_j)$, then $d_i = d_j$.
- **Termination:** If all honest parties give an input of the form $(\textsc{Vote}, \mathsf{baid}, \cdot)$, then eventually all honest parties give an output of the form $(\textsc{Decision}, \mathsf{baid}, \cdot)$.

Figure 9.7 An ideal functionality ABA for Asynchronous Byzantine Agreement.

9.5 Defining Asynchronous Byzantine Agreement

We now move on to study asynchronous Byzantine agreement. We define it via the ideal functionality ABA in Fig. 9.7. We define it using the pattern of trace-property ideal functionalities, where it is the adversary decision the output under some restrictions given by trace properties. We will not discuss it further, the description should be self-explanatory given the discussion on how to define synchronous Byzantine agreement and synchronous Broadcast.

9.6 Implementing Asynchronous Byzantine Agreement using Coin-Flip

The general idea behind implementing asynchronous Byzantine agreement is to try to stabilice the network in the sense that all honest parties have the same vote. This is done by running a sequence of protocol which update the votes. Crucially, these protocols all have the property that if the network is stable, then it will remain stable on the same value. Some of the protocols will also have the property that even if the network is not stable, they will stabilize it with some non-zero probability. If you run enough of these, then eventually the network will stabilize. Some of the protocols will have the property that they can detect whether the network is stable. This will help in knowing when to terminate. Finally we will discuss how to terminate gracefully in an synchronous protocol once it has been detected that the network is stable.

9.6.1 Vote Updating Protocols

Definition 9.2 (stability protocols) We call an asynchronous protocol a vote updating protocol (VUP) is it has the following property. At the beginning of the protocol each honest party has a vote $V_i \in \{0, 1\}$. The protocol should be terminating in the usual sense that if all honest parties start running the protocol, then eventually all honest parties will terminate with an updates vote $W_i \in \{0, 1\}$. We call the inputs to the protocol stable on V if all honest parties have $V_i = V$. We call them stable if they are stable on some bit V. We use the same terminology for the outputs. We call a vote updating protocol stability preserving if it holds that when the inputs are stable on V, then the outputs will always be stable on V too. We call a VUP a σ-stabilizer if it holds for all configurations of the inputs that the output will be stable with probability at least σ. We call a VUP a detector if outputs to each party a truth value $D_i \in \{\perp, \top\}$ with the property that if the input is stable, then $D_i = \top$ for all honest parties and that if $D_i = \top$ for any honest parties, then the output is stable.

Below we will describe several vote updating protocols using signaturs. We will be a bit informal in how we describe the use of signatures to be able to focus on the important parts. We assume the parties have access to toyPKI to learn each others public keys. We also assume signatures on different rounds and protocols will sign some unique strings along with the votes to avoid replay attacks where signatures from honest parties in one protocol or round are reused on some other protocol of round.

Each P_i runs:

1. Send V_i to all parties along with a signature on V_i.
2. Collect $n - t$ signatures on bits.
3. Pick a bit $J_i \in \{0, 1\}$ for which there are $t + 1$ distinct signatures.[a]
4. Send the value J_i to all parties along with the $t + 1$ signatures. We call J_i a justified vote.[b]
5. Collect $n - t$ justified votes.
6. If all $n - t$ justified votes are on the same J then let $W_i = J$. We call this a hard vote
7. Otherwise pick $W_i \in \{0, 1\}$ in any other way. We call this a soft vote. For the same of argument, say we ask the adversary for W_i in this case.

[a] This is possbile as there are $n - t = 2t + 1$ signed votes collected and they are all on some bit, 0 or 1, so one of the bits must have gotten $t + 1$ votes.

[b] To be a justified vote the $t + 1$ signatures need to be correct. The receiver will have to check this.

Figure 9.8 Basic Vote Updating Protocol BaVUP for $n = 3t + 1$ parties of which at most t are Byzantine corrupted

In Fig. 9.8 we describe the basic VUP. It is clear that the protocol is terminating as it always waits for at most $n - t$ values and then goes to the next step. We argue that the protocol is stability preserving. If the input is stable on V, then no honest parties will vote for the other bit $V' = 1 - V$. Therefore V' will get at most t signatures (from the corrupted parties). Therefore V' cannot become a justified value. Therefore, if a party collects $n - t$ justified values J_j, then they are all for V which implies that all honest parties will hard vote$W_i = V$. Notice that this proof holds no matter how W_i is picked when we ask the adversary to pick it. This gives us a lot of freedom in how to soft vote when we start massaging BaVUP below.

Each P_i runs:

1. Send V_i to all parties along with a signature on V_i.
2. Collect $n - t$ signatures on bits.
3. Pick a bit $J_i \in \{0, 1\}$ for which there are $t + 1$ distinct signatures.
4. ACast the justified vote J_i to all parties along with the $t + 1$ signatures.
5. Collect $n - t$ justified votes.
6. If all $n - t$ justified votes are on the same J then hard vote $W_i = J$.
7. Otherwise pick the soft vote W_i to be uniformly random.

Figure 9.9 Stabilizing Vote Updating Protocol SVUP for $n = 3t + 1$ parties of which at most t are Byzantine corrupted

9.6.2 Stabilizing VUP

We can use the basic VUP to produce more complicated VUPs. We now describe a Stabilizing VUP in Fig. 9.9. It is like BaVUP with the change that it will use ACast to send the justified value and it picks soft votes uniformly at random. Using randomness form SVUPs was first proposed by Michael Ben-Or[1].

The reason why we use ACast to send the justified values is to be able to prove the SVUP Lemma which is crucial for stabilization.

Lemma 9.3 (SVUP) *In* SVUP *it holds that if an honest party saw $t + 1$ justified votes for the bit J, then no honest party gave or will give a hard vote for $W \neq J$.*

PROOF Assume sone honest party saw $t + 1$ justified votes for the bit J. Since we ACast the justified votes J_i even the corrupted parties have to send the same value to all honest parties. Therefore, if an honest party sees $t + 1$ justified votes for J, then all honest parties will see at least 1 justified votes for J. Namely, they collect $n - t$ votes, so there are at most t values they do not collect, so they must see one of the $t + 1$ votes for J. If an honest party sees a justified vote for J then they do not do a hard vote for $J' \neq J$. □

We can now prove that SVUP is a stabilizing VUP. Before we do this, we want to add that the a stabilization probability of SVUP is poor. If $n = 10$, then the probability of stabilization is about one permille. It is possible to make efficient stabilizing VUPs with much better parameters (for instance 50%), but we will not explore this here. We return to this in Chapter **??**.

Theorem 9.4 SVUP *is a σ-stabilizer VUP for $\sigma = 2^{1-n}$.*

PROOF Since SVUP is based on BaVUP It is clearly still terminating and stability preserving. We now argue that it is a σ-stabilizer for $\sigma = 2^{1-n}$. To see this, assume that no honest party has a hard vote. Then they will all pick W_i uniformly at random, so they will be equal with probability at least 2^{1-n}. Assume then that some honest party in fact has a hard vote. Say that the hard vote is for W. We want to argue that also in this case it holds that all honest parties will have output W_i with probability at least σ. We need to argue two things. First of all we need to argue that there is no hard vote for $W' = 1 - W$. This is true and the reader should be able to prove this by now. The second thing we need to argue is that all soft votes happen to become W with probability at least σ. This is not as easy as it seems as it could be that the soft vote is done *before* the hard vote and that the adversary can somehow steer the network to make sure the hard vote is always on the opposite value of the first random soft vote. What we need to argue is therefore that at the time where the first honest party makes a soft vote there is at most one value V on which any party might make a hard vote. To see this assume that P_i is about to do a soft vote. At this time P_i received $n - t$ justified votes. Since $n - t \geq 2t + 1$ it follows that at least $t + 1$ of these soft votes are on the same value J. It then follows from the SVUP Lemma that J is then the only value on which there can ever be a hard vote. So if all soft votes happen to be for J, then all soft votes and hard votes will be identical. And since

J is fixed before the soft votes are picked at random, each of the has probability $1/2$ of hitting W. □

Exercise 9.2 (Agreement on Hard Votes in SVUP) Prove that in BaVUP and SVUP if two honest parties P_i and P_j have hard votes on W_i and W_j, then $W_i = W_j$.

Each P_i runs:

1. Send V_i to all parties along with a signature on V_i.
2. Collect $n - t$ signatures on bits.
3. Pick a bit $J_i \in \{0, 1\}$ for which there are $t + 1$ distinct signatures.
4. ACast the justified vote J_i to all parties along with the $t + 1$ signatures.
5. Collect $n - t$ justified votes.
6. If all $n - t$ justified votes are on the same J then hard vote $W_i = J$. Let $D_i = \top$.
7. Otherwise pick the soft vote W_i to be the W that got the most justified votes. Let $D_i = \bot$.

Figure 9.10 Detecting Vote Updating Protocol DeVUP for $n = 3t + 1$ parties of which at most t are Byzantine corrupted

9.6.3 Detecting VUP

We can also massage BaVUP into a detecting VUP DeVUP, see Fig. 9.10.

Theorem 9.5 DeVUP *is a detecting VUP.*

Proof Again termination and stability preservation follows from the fact that we used BaVUP as the basis. We argue stabilization. Assume the input is stable on V. Then only V can get a justified vote. Therefore all honest parties will get $n - t$ justified votes for V and set $D_i = \top$. Assume then that any honest party sets $D_i = \top$. Then it saw $n - t = 2t + 1$ justified votes for some value V. All other honest parties collect $n - t$ justified votes. So there are at most t of the values that P_i saw that they will not see. So they will see at least $(2t + 1) - t = t + 1$ justified votes for V. Since they collect only $2t + 1$ votes it follows that V will be the value getting more justified votes. Hence all honest party will (at least soft) vote for V too. □

9.6.4 Time Capsules

Notice now that if you would run a protocol where you run an infinite loop of first SVUP and then DeVUP, the network will eventually stabilize on some value V and via DeVUP all honest parties will eventually learn this. All we need is therefore a way to determine when we stop and output V. However, termination in asynchronous protocols is harder than it might seem at first. The reason is that when a party terminates, it should also stop participating in all sub-protocols that it started. This is typically needed to get the type of termination we want, namely that all resources allocated to a terminate process are now free. Consider for instance a party P_i which has started an ACast as a sub-protocol. If P_i terminates without terminating the broadcast, then it might have to let a process run forever that participates in the broadcast. But if you have to allocate a process that runs forever, you did not really terminate, the allocated process is still occupying system resources. On the other hand, terminating all sub-protocols might be dangerous. Recall that some slow correct processes P_j might be arbitrarily behind: it might still not have sent or received its first message when P_i terminates. If P_i terminates before P_j "wakes up", then P_i will not be around to participate in the sub-protocol that would allow P_j to progress, it would for instance not be present to participate in the Bracha broadcast that implements the ACast that P_j should initiate and therefore P_j might not be able to broadcast its messages.

The main trick in the book to solve this problem is that when honest parties terminate and shut down, then they send messages that will later inform slow correct processes about the result of the computation when they wake up. We will call these time capsule messages: they are sent now before P_i terminates and they will be around to be picked up by P_j when it wakes up. For this to work it is important that these time capsule messages get delivered even after P_i terminated. Therefore we will make one exception to killing sub-protocols when one terminates:

Definition 9.6 (Asynchronous Termination) When we say that a party terminates, we mean that it terminates its own process, plus its processes in any sub-protocol it started. There is one important exception: it will forever keep running the process that implements the authenticated channels it is using.

Implementing this in practice could be done by having the operating system of the computer handle the sending and receiving of messages, or delegate it to some permanent daemon process. When local processes send a message, they hand it to the daemon, who will ensure to deliver them even after the process shuts down.

An important fact about termination is that if a correct processes terminates, then it can affect the liveness of any sub-protocol it participated in. However, it *cannot* affect any safety property of a sub-protocol it participated in. Namely, the party terminating is no different from all its future messages being arbitrarily delayed. And delaying the messages of honest parties cannot affect safety properties, as this is a possible behaviour of the network. We will use this fact below, sometimes implicitly. But we make sure to use it explicitly once to show how the fact is used.

Besides being left behind for slow processes, the time capsule messages can sometimes be used to even allow reasonably up to date processes to terminate faster. It might in these cases be a good idea to make sure they arrive quickly to all parties. To make sure the time capsules reach all parties quickly so they can shut down the main protocol as fast as possible, one can use piggy bagging where the time capsule messages are sent along with the next message in the main protocol.

Input: P_i: On input $(\textsc{Vote}, \mathsf{baid}, v_i)$ Let $V_i^0 = v_i$. Let $r = 0$. Let $\mathsf{GaveOutput} = \bot$. Then start Stabilize and Output.

Stabilize: P_i: Run this loop forever:

1. Run SVUP with input V_i^r to get W_i^r.
2. Run DeVUP with input W_i^r to get V_i^{r+1} and D_i^r.
3. If $D_i^r = \top$, then output $(\textsc{Decision}, \mathsf{baid}, V_i^{r+1})$, let $\mathsf{GaveOutput} = \top$ and send $(\textsc{TimeCap}, V_i^{r+1})$ [a] to all parties.
4. Let $r \leftarrow r + 1$ and go to step 1.

Output: P_i:

Term 1 On receiving $(\textsc{TimeCap}, d)$ from $t+1$ parties, send $(\textsc{TimeCap}, d)$[b] to all parties. If $\mathsf{GaveOutput} = \bot$, then set $\mathsf{GaveOutput} = \top$ and output $(\textsc{Decision}, \mathsf{baid}, d)$.

Term 2 On receiving $(\textsc{TimeCap}, d)$ from $2t + 1$ parties, terminate.

[a] This is a time capsule message, it is delivered even if the sender terminates VUPABA.
[b] This is a time capsule message.

Figure 9.11 A VUP based protocol for Asynchronous Byzantine Agreement called VUPABA. It works for $n = 3t + 1$ parties of which t can be Byzantine corrupted.

9.6.5 ABA from VUPs and Time Capsules

We now show in Fig. 9.11 how to implement ABA given SVUP, DeVUP and a time capsule add-on. The intuition behind the protocol is that when a party P_i sees that $D_i^r = \top$, then it knows that the network is stable and that all other parties will realise it no later than in the next round. However, it keeps running to help most of the other reach the next round. When enough reached the next round, ensuring that enough time capsules are around, P_i can terminate. The logic behind the two thresholds in the shutdown mechanism is the same as in Bracha Broadcast.

Theorem 9.7 *The protocol* VUPABA *securely implements* ABA *for* $n = 3t + 1$ *parties out of which at most* t *are Byzantine corrupted.*

Proof

Termination

We first argue that if no correct P_i ever terminates, then eventually some correct P_i terminates, which is of course a contradiction. So we can use this to conclude that eventually some correct P_i terminates. That sounds as a very backwards way to argue, but it is a common way to do it as it allows us to assume throughout the proof that all correct parties keep progressing, which in turn allow us to assume that all sub-protocol has also their liveness properties.

So, assume that no correct P_i ever terminates. Since no parties ever waits for more than $n - t$ parties, all correct parties eventually progress to higher and higher round numbers r. By the stabilization property of SVUP and the stabilization preservation of SVUP and DeVUP the network will eventually styabilize and DeVUP will detect this giving all honest parties output $D_i^r = \top$. Say the network stabilises in round r on value V. Since no correct process ever terminates, they will all eventually reach round $r + 1$. Since the network is stable, they will all vote V. Therefore they will all get the output $D_i^{r+1} = \top$. Since no correct process ever terminates, they will all eventually send and receive $(\textsc{TimeCap}, d)$, and then terminate. Contradiction.

So we know that it cannot be true that all honest parties run forever. So we know that eventually some correct P_i terminates. When it did terminate, it saw $2t + 1$ messages of the form $(\textsc{TimeCap}, d)$. At least $t + 1$ of these were sent by correct parties. When these $t + 1$ correct parties sent the messages received by P_i they sent the message to all honest parties. Therefore all honest parties that do not terminate for other reasons will eventually receive $t + 1$ message of the form $(\textsc{TimeCap}, d)$ and then they will send $(\textsc{TimeCap}, d)$. To argue this we are using that all messages that are sent or also delivered, even after the sender terminated. The take-away message here is that all correct processes that stay alive long enough after P_i terminates will eventually send $(\textsc{TimeCap}, d)$.

We then argue that it is actually impossible for an honest party to terminate without sending $(\textsc{TimeCap}, d)$. Namely, to terminate P_j saw $2t + 1$ messages $(\textsc{TimeCap}, d')$, and when some correct party sends $(\textsc{TimeCap}, d)$ then no correct party sends $(\textsc{TimeCap}, d')$ for $d' \neq d$, so P_j saw $2t + 1$ messages $(\textsc{TimeCap}, d)$. But then it of course also saw $t + 1$ messages $(\textsc{TimeCap}, d)$ and hence it sent $(\textsc{TimeCap}, d)$.

So, when some honest P_i terminates, then all honest P_j will eventually send $(\textsc{TimeCap}, d)$. Therefore all honest P_k will eventually receive $n - t \geq 2t + 1$ such messages and terminate.

Agreement

Assume that P_i outputs $(\textsc{Decision}, \mathsf{baid}, d)$. It can do this in two places, either in Step 3 or in *Term 1*. If it does it in *Term 1*, then it saw at least one $(\textsc{TimeCap}, d)$ sent by a correct party. This means that at some point some honest process sent $(\textsc{TimeCap}, d)$ in Step 3, so at some point some P_i output $(\textsc{Decision}, \mathsf{baid}, d)$ in Step 3. So it is enough to argue that if some correct P_i outputs $(\textsc{Decision}, \mathsf{baid}, d)$ in Step 3 and some correct P_j outputs $(\textsc{Decision}, \mathsf{baid}, d')$ in Step 3, then $d = d'$. So, assume that a correct P_i outputs $(\textsc{Decision}, \mathsf{baid}, d)$ in Step 3 in round r_i and some correct P_j outputs $(\textsc{Decision}, \mathsf{baid}, d')$ in Step 3 in round r_j and that $r_i \leq r_j$. When P_i output $(\textsc{Decision}, \mathsf{baid}, d)$ in Step 3 it had $D_i^{r_i} = \top$ and $V_i^{r_i+1} = d$. This means that the network was stable and therefore $V_j^{r+1} = d$ for all $r \geq r_i$ and all correct P_j that reach round r_j. Since $r_j \geq r_i$ it follows that $d' = V_j^{r_j+1} = d$.

Notice that when we conclude from $D_i^{r_i} = \top$ and $V_i^{r_i+1} = d$ that $V_j^{r+1} = d$ for all $r \geq r_i$ and all correct P_j we use stability preservation of SVUP and DeVUP only. This is correct even if at this point in the protocol some processes terminated already as it is a safety property and therefore holds even if some correct processes terminated.

Validity

Assume that some correct P_i gives an output $(\textsc{Decision}, \mathsf{baid}, d_i)$. As we saw above, this means that at some point some correct P_j output $(\textsc{Decision}, \mathsf{baid}, d_j)$ in Step 3. Assume without loss of generality that P_j is the first party to do this. At this point the network was stable on d_j. Let $d = d_j$. The network will remain stable on d forever after, as stability only relies on safety properties. Hence any other correct party P_k outputting $(\textsc{Decision}, \mathsf{baid}, d_k)$ in Step 3 will have $d_k = d$. Therefore no correct P_i will ever send $(\textsc{TimeCap}, d')$ for $d' \neq d$. Therefore no correct party gets $t + 1$ messages $(\textsc{TimeCap}, d')$ for $d' \neq d$. Therefore no correct party outputs $(\textsc{Decision}, \mathsf{baid}, d')$ for $d' \neq d$ in *Term 1*. It is there sufficient to argue validity of d_j is it was output as $(\textsc{Decision}, \mathsf{baid}, d_j)$ in Step 3. But in Step 3 the network is stable in d_j. So by stability preservation of SVUP and DeVUP, if all honest parties had the same input d to VUPABA then this was preserved and $d_j = d$. If all honest parties did not have the same input, then all outputs are valid. □

Notice that it is not crucial to VUPABA the the sequence of VUPs is exactly SVUP, DeVUP, SVUP, ... Any sequence will do as long as it always has another SVUP coming up at some point and another DeVUP coming up at some point. One can use this to optimize for some special case that occur often in practice, like many honest parties or highly biased inputs.

Exercise 9.3 (Better VUPABA) Make a version of VUPABA with the property that if there are $n - t$ honest parties with the same input vote, then the protocol is be guaranteed to shut down without even having completed on SVUP. Consider using piggy bagging.

9.6.6 Interactive Coin-Flips

The above SVUP uses a local coin flip. To get better probability that all honest parties get the same coin, one need to do something smarter where the parties interact. One would then run the coin flip protocol right after SVUP and use the outcome to set the soft vote. All that is needed for stabilization is that often enough the coin is random and identical at all parties. The ideal functionality of a coin flip protocol would be something along the lines of the following protocol.

Definition 9.8 (Coin-Flip (informal)) Coin-Flip is given by an ideal functionality Flip. Party P_i has a port Flip_i. Each coin flip will have a unique flip ID fid. Honest parties input $(\textsc{Flip}, \mathsf{fid})$ to start the coin flip. At some point Flip outputs $(\textsc{Coin}, \mathsf{fid}, c_i \in \{0, 1\})$ to P_i to inform it that the outcome was c_i. It is allowed that $c_i \neq c_j$ and that the c_i's were picked by the adversary. But it must always eventually happen that a coin-flip is good. A good coin-flip is one in which that after the first honest party input $(\textsc{Flip}, \mathsf{fid})$ the IF generated a uniformly random $c \in \{0,1\}$ and set $c_i = c$ for all honest P_i.

9.7 Impossibility of Deterministic Asynchronous Byzantine Agreement*

The stabilizing VUP above used randomness. It turns out that this is needed. The reason is that if we could have made a deterministic version of SVUP, then VUPABA would be deterministic. And then we would have a deterministic protocol solving ABA. And it unfortunately turns out that whomever created the universe decided that it should be impossible to solve ABA without using random choices. This might sound odd, but we can actually prove that this is the case. We do this now. It is an example of a so-called impossibility result. They are very hard to find as one have to prove that out of all the infinitely many protocol that exists none of them solves the problem at hand. We can go over all the infinitely many protocols. So instead we use a proof by contradiction. We assume that we have a protocol that solve the problem. Then we show that it has an impossible property, ruling out that it exists.

We look at so-called deterministic protocols with the following property. Let s be a scheduling sequence, i.e., a sequence of commands of the form $c_1, c_2, c_2, \ldots$, where each c_i is a command executed while running the protocol, like "deliver the message with message identifier *mid*", "crash party P_i", or "schedule party P_j". We look at protocols where if all inputs are fixed and the scheduling sequence c is fixed. Then so is the output of all parties.

For simplicity, we look at the proof for $n = 4$ parties, but it holds for any number of parties. Each P_i has as input a vote v_i. The vector of inputs is denoted by $\mathbf{v} = (v_1, v_2, v_3, v_4)$. Take any protocol achieving Byzantine agreement for four parties. For an input $\mathbf{v}$ let $V(\mathbf{v}) \subseteq \{0,1\}$ be the set of values that the parties might output when run on those inputs. There might be more than one such value, as the output might depend on who is crashed and in which order messages are scheduled. We know that $V(\mathbf{v}) \neq \emptyset$ as the protocol always terminates with some output. By validity we know that

$$V(0,0,0,0) = \{0\}$$

and

$$V(1,1,1,1) = \{1\} .$$

Now we ask what the V-values could be for the other inputs below:

$$V(0,0,0,0) = \{0\}$$

$$V(1,0,0,0) = ?$$

$$V(1,1,0,0) = ?$$

$$V(1,1,1,0) = ?$$

$$V(1,1,1,1) = \{1\} \ .$$

We argue that one of them must be $\{0,1\}$. Assume that this was not the case, i.e., all the sets have size one. Then at some point it has to change from $\{0\}$ to $\{1\}$. Assume without loss of generality that it is here:

$$V(1,1,0,0) = \{0\}$$

$$V(1,1,1,0) = \{1\}$$

Now consider the following execution of π on input $(1,1,0,0)$. In the first step, crash P_3. Then run according to any message schedule. Since the assumption is that $V(1,1,0,0) = \{0\}$, then the output in this scenario can only be 0. Now consider the following execution of π on input $(1,1,1,0)$. In the first step, crash P_3. Then run according to the same message schedule. Since the assumption is that $V(1,1,1,0) = \{1\}$, then the output in this scenario can only be 1. But this is in contradiction! The only difference between the executions are the input of P_3, but we crash P_3 before it can send any messages, so the output cannot depend on the input of P_3.

Good, so there must be an input that gives V-value $\{0,1\}$. Assume without loss of generality that it is

$$V(1,1,0,0) = \{0,1\} \ .$$

Now let s be any scheduling sequence, i.e., for some number of steps in the protocol, it says which message to deliver next. Let

$$V(1,1,0,0,s) \subseteq \{0,1\}$$

be the values the protocol can output after first running according to s and then, if all commands in s have been executed, doing any scheduling from then on. Since the protocol terminates, we know that for all sufficiently long sequences s it holds that

$$V(1,1,0,0,s) = \{0\}$$

or

$$V(1,1,0,0,s) = \{1\} \ .$$

We also know that

$$V(1,1,0,0,\epsilon) = \{0,1\}$$

for the empty sequence ϵ. So there exists a sequence s of maximal length such that

$$V(1,1,0,0,s) = \{0,1\} \ .$$

What it means for s to be of maximal length is that no matter which deliver command d we append to s, it will hold that

$$V(1,1,0,0,(s,d)) = \{0\}$$

or

$$V(1,1,0,0,(s,d)) = \{1\} .$$

Even better, since

$$V(1,1,0,0,s) = \{0,1\} ,$$

there must exist deliveries d_0 and d_1 such that

$$V(1,1,0,0,(s,d_0)) = \{0\}$$

or

$$V(1,1,0,0,(s,d_1)) = \{1\} .$$

Assume first that d_0 is a delivery to P_i and d_1 is a delivery to P_j. Then first executing d_0 and then d_1 or the other way around will have the same effect, so

$$V(1,1,0,0,(s,d_0,d_1)) = V(1,1,0,0,(s,d_1,d_0)) .$$

But we know from above that

$$V(1,1,0,0,(s,d_0,d_1)) = \{0\}$$

and

$$V(1,1,0,0,(s,d_1,d_0)) = \{1\} ,$$

a clear contradiction.

So the only remaining possibility is that that d_0 and d_1 are deliveries to the same P_i. This would mean that the output is decided from which message gets delivered first to P_i. However, this cannot work: if P_i crashes before he can tell anyone what happened, the rest of the network cannot make the right decision. More formally, let c_i be a scheduling which starts by crashing P_i and never delivers a message from P_i to any other party. Clearly

$$V(1,1,0,0,(s,d_0,c_i)) = V(1,1,0,0,(s,d_1,c_i)) ,$$

as the rest of the network will never hear which message P_i received before crashing. But we know from above that

$$V(1,1,0,0,(s,d_0,c_i)) = \{0\}$$

and

$$V(1,1,0,0,(s,d_1,c_i)) = \{1\} ,$$

a clear contradiction.

So that is the bad news. In an asynchronous network with the possibility of a single crash, a single step can never help make a decision as the rest of the network might not hear about that step. Therefore a protocol cannot go from an undecided state to a decided state.

Not all hope is lost, however. If we look carefully at the proof, we see that we needed to very carefully find the point where the deciding message was delivered and then crash the party getting it. For this it is actually important that the protocol is deterministic. If the parties would somehow make decisions based on internal randomness, like draw a random bit b and send it like SVUP did, then the outcome of the protocol would not be fixed by the scheduling, and then the above argument breaks down.

10

Key Management and Infrastructures (DRAFT)

Contents

10.1 Introduction to Key Management

So far, we have only looked at what can be done assuming users have access to the keys they need, and we have ignored questions about how you can get these keys in place, manage and update them. When solving such problems, a first thing to be aware of is the following basic fact of life:

Any secret system parameter runs a greater risk of being revealed, the longer time you keep it constant, and the more you use it.

This is why we have to assume that an adversary knows the algorithms and protocols we are running (commonly known as *Kerckhoffs's principle*), and why we cannot assume that the encrypted messages we send will remain secret for ever. It is also the reason why it is good practice to change the keys you use at regular intervals.

10.2 Two-party Communication

The standard solution for achieving confidentiality (or authenticity, or both) in two-party communication between A and B (if we use secret-key technology) is to have a long-term key K_{AB} agreed between A and B initially, used only for sending keys from A to B. Then to send a message M, A will generate a random so called session key K, and will send $E_{K_{AB}}(K), E_k(M)$ to B. It should be evident how B can recover the message.[1] In real life K will often be reused for several messages, but will be deleted after a short time, typically as soon as the connection is closed.

Of course, we do have a long-lived key here, namely K_{AB}, but generally speaking, this cannot be avoided – although of course even K_{AB} should be changed at regular (longer) intervals. The argument for the two-level system is that if we used K_{AB} directly on data, this would give an adversary a very large amount of data encrypted under one key, and this generally makes cryptanalysis easier. With a multi-level system of keys, we can make sure that every key is used only on a limited amount of data.

If we were to use a similar idea for a multi-user network, we would soon run into trouble: every user would have to store a key for every other user he wants to speak privately with. This can be difficult, or even impossible in large, dynamic groups of users. In any case it would mean that the total number of keys one needs to manage would grow quadratically with number of users because there are $n(n-1)/2$ pairs if we have n users in total. This gets quite impractical when n is large.

To solve this problem, there are two basically different approaches, namely Key Distribution Centers and Certification Authorities.

10.3 Key Distribution Centers (KDC)

KDC's are based only on secret-key technology. The basic principle is that we have one KDC and many users, and every user A shares a key K_A with the KDC. When A wants to talk to B, we have the KDC generate a key K for the session, and send $E_{K_A}(K)$ to A and $E_{K_B}(K)$ to B. Then both parties can recover K and communicate using this key. There is an almost infinite number of variations on this theme, but the basic principle remains the same. One real-life example of such a system is Kerberos, which originated in some US universities and is also used in some companies. It contains many elements not described here. For instance, the simplistic solution sketched above does not allow A to verify that a secure channel has actually been established with B, or that B is even active at the other side. We look at this so-called mutual authentication problem in more detail later.

As mentioned above, there are KDC-like solutions out there, but they are not all that common, because of the single point of failure which the KDC represents. In particular:

[1] This is just a simplistic example protocol to illustrate the principle – for instance there are no provisions for authenticity here, meaning that B cannot verify he is talking to A.

- all users must trust the KDC completely, since the KDC can decrypt or forge all traffic if it decides to misuse its knowledge.
- if the KDC is down, the entire secure communication systems becomes unavailable.

Hence one often prefers a public-key based solution as we describe in the next section.

10.4 Certification Authorities (CA)

We can first remark that in the simple two-player example above, the shared key K_{AB} could be replaced by for instance a public key pair (sk_B, pk_B) generated by B where only B knows sk_B and pk_B is public. A can then send $E_{pk_B}(K)$ to B where K is a session key. As mentioned before, this means that no secret key needs to be shared initially, but we still need to ensure that A uses the correct public key. It is not reasonable to assume that every user has an updated and guaranteed-to-be authentic list of everyone else's public key. Remember, the *public* in public key means that the system is secure even if the adversary has access to this key, not that all honest parties necessarily know and agree on the value of this key!

So we make instead a less demanding assumption: assume we have an entity we call a Certification Authority (CA), with its own key pair (sk_{CA}, pk_{CA}). And assume that we can ensure that all users get an authentic copy of pk_{CA}. We can then do the following: each user A in the system must contact the CA, identify himself in some way, and send his public key pk_A to the CA. It is important to understand that the way A identifies himself cannot be via cryptography only: before A has registered, A and the CA do not share any secret keys, and the CA does not know any public key it can safely assume belongs to A. Therefore, the CA must verify A's identity is some non-cryptographic "physical" way, and it must be able to verify that the public key it gets, actually came from the person it identified. In practice, this takes many forms. In some (rare) cases, A must show up in person, in other cases, a pin code is sent to A in paper mail and used later to log into the website of the CA.

If the CA accepts the identity of the user, it will issue a so called *certificate*, which consists of a string ID_A identifying A uniquely, the public key pk_A, and the CA's signature $S_{sk_{CA}}(ID_A, pk_A)$ on the former two pieces of data. This can be thought of as a kind of receipt for having registered - the CA certifies that it identified the user as being A, and that pk_A was the key that was presented to it. Because all users were assumed to have pk_{CA}, everyone can check the CA's signature, and so users can communicate securely if they first exchange certificates.

The primary difference to the KDC solution is that the CA is trusted in a different way: we trust it to not issue certificates on false grounds, but it never sees any *secret* keys used for communication. Also once certificates are issued, it is possible to set up a secure communication without necessarily involving the CA.

In real life, things are not as simple, however: if a user suspects that his private key has been compromised, it should possible for him to report this, and have his certificate revoked, much like credit cards can be reported stolen. This means that there must also be an option to check whether a given certificate is still valid. In other words, the CA should implement an on-line service for this purpose, or be able to publish black-lists of revoked certificates. This is one reason why certificates should also contain a validity period as part of what is signed by the CA, since if a certificate never expires, black lists would grow indefinitely. In general, certificates may contain all sorts of other control information, so a certificate is typically a rather long data record with for instance the following fields:

- Name of the certificate owner
- Name of the CA who signed the certificate
- Method used by CA to check identity of certificate owner
- Date issued
- Validity period
- Rights and privileges of certificate owner.
- Crypto algorithm to be used for checking this certificate
- Crypto algorithm used by certificate owner
- Public key of certificate owner
- Signature of CA.

Also, there is of course more than one CA in the world. In the US, companies like Verisign and AT&T issue certificates, in Denmark for instance DanID offer such services. So it will often be necessary to have methods for handling the situation where two users do not have certificates from the same CA. One way to do this is if the two CA's certify each other's public keys. Some notation to describe this: a certificate issued by certification authority CA to user A with public key pk is denoted $Cert_{CA}(A, pk)$.

Now, say user A has a certificate from CA_1 and user B has one from CA_2. Now A receives $Cert_{CA_2}(B, pk_B)$, which he cannot directly check because he does not have CA_2's public key. He does have CA_1's public key, however, so if he also gets a certificate $Cert_{CA_1}(CA_2, pk_{CA_2})$, then A can first verify this certificate, which convinces him that the public key of CA_2 really is pk_{CA_2}. He can now verify $Cert_{CA_2}(B, pk_B)$ to get convinced about pk_B.

A more general way is to use so called *certificate chains*: such a chain is an ordered list of certificates, of the following form

$$Cert_{CA_1}(B, pk_B), Cert_{CA_2}(CA_1, pk_{CA_1}), ..., Cert_{CA_n}(CA_{n-1}, pk_{CA_{n-1}})$$

where the first entry is a certificate where CA_1 certifies the public key of the user (B) in question. In the second entry, CA_2 certifies the public key of CA_1, etc. until the last entry where CA_n certifies the public key of CA_{n-1}. If another user A is able to collect enough certificates so he can form a chain as above, where he knows already the public key of CA_n, then he can verify the entire chain and hence also the public key of B.

It is important to understand, however, two limitations of certificate chains:

A should only trust the end point (B's public key) to the extent that he trusts that *every* CA involved in the chain has not issued fake certificates (and this has happened in practice – CAs can be hacked, their keys can be stolen, etc.). Second, certificate chains do not remove the need for at least one public key that must be known to users initially, in particular, verifying a certificate chain cannot be done unless one knows beforehand an authentic copy of pk_{CA_n}.

In a practical situation, the way in which this is ensured is usually that the required public key(s) are delivered to the user together with the software needed to generate keys and do encryption and signatures. For example, standard browsers like Chrome, Mozilla Firefox, Safari, etc., come with a set of preinstalled *root certificates*. These are in fact simply built-in public keys of a number of CA's. Although such a so called *root certificate* has the data format of a certificate and contains a signature, this signature was done by the CA itself, i.e., the certificate is self-signed. Therefore, that signature does not prove anything in itself – the *only* trust one can place in a public key obtained this way is the trust that follows from having received it from a trusted source. Note that trust in such a key can also be obtained in other ways than having it be preinstalled in software. For instance, a hash value (a fingerprint) $h(pk)$ of the public key pk could be sent to you in paper mail. Then you type the hash value in yourself on your machine, so that when pk is sent on the net, your software can verify that $h(pk)$ matches the value you typed.

In order to have interoperability between different software products handling certificates, a number of international standards have been defined that specify which elements a certificate should contain, in which order they should appear, how the data should be formatted etc. By far the most common standard for certificates is the X.509, which comes originally from the international association of telecom companies, CCITT. This standard is the only one in use on the Internet, and virtually any browser capable of handling secure communication knows how to handle X.509 formatted certificates. Unfortunately, X.509 is not very flexible, and has therefore been revised several times, each time allowing new extensions to the types of information that a certificate can contain. It is therefore not always guaranteed that programs can communicate, even if they both claim to understand X.509.

Finally, while up to few years ago it was relatively expensive and complicated for small organizations to get a certificate from one of the existing commercial certificate authorities, the situation changed when a number of profit and non-profit organizations started *Let's Encrypt*, a free certificate authority with the goal to push encrypted communication as the default setting for the entire world wide web.

10.5 Limitations on Key Management

We have seen that systems for key management often tend to be hierarchic: one key is protected from modification or leakage by another key, which is again protected by a third key, etc. Sometimes the structure is not a simple tree structure, and one may have keys that are protected by different keys at different locations. But any such structure must of course have an end: one or more keys that are not protected by other keys, and so can only be protected by non-cryptographic means. This is for instance true for the public key of the last CA in a certificate chain, and also for the PIN code that protects access to a user's private key on his machine. This leads to the following, very important conclusion:

Any secure system using cryptography must make use of one or more keys that are protected only by physical, non-cryptographic means

And so a real-life analysis of any such system cannot be limited to cryptography alone. This is the main motivation for the next three subsections, on password- and hardware-security and biometrics. These three sections focus on how a system can securely identify a user who wants access to some resource, such as a key. A user can be identified by something he *knows* (a password), something he *has* (some hardware) or something he *is* (biometrics).

After this, we look at how one can make sure that the resource can only be accessed by going through the normal system check.

10.6 Password Security

Passwords or PIN codes are an essential part of security in many systems, not just as a protection mechanism protecting access to cryptographic keys, in fact they are often the only security mechanism employed. Unfortunately, they are often the weakest link in the chain because they have to be remembered by humans. This often has the effect that people either choose passwords that are easy to guess, or write them down, which of course degrades security in either case.

In the following we will look at the general case where a password user and a password verifier both know a password specific to the particular password user. The idea is that the user will somehow present the password to the verifier to prove his identity, so we want to ensure as well as we can that no third party gets to know the password. There are (at least) 4 important security aspects of password security, and corresponding to these, there are 4 types of attacks we must protect against:

- How is the password chosen – Can the adversary guess the password and verify his guess?
- How is the password transmitted between the password user and verifier – Can the adversary get hold of the password while it is in transit?
- How is the password stored by the password user – Can the adversary steal the password from the user?
- How is the password stored by the password verifier – Can the adversary steal the password from the verifier?

10.6.1 Choosing and guessing Passwords

As for the choice of passwords, it is clear that from a security point of view, we want that they are selected from as large a set as possible, which implies that long passwords with a large number of choices for each character are better. For instance, there are 10.000 different PIN codes for a DanKort, but the number of different UNIX passwords is several billions times larger, about 2^{52}. In general, if the password is chosen from a character set of size C and has length ℓ, then there are C^ℓ possible passwords. This number grows much faster as a function of ℓ than as a function of C, so if one wants a large number of possibilities, then increasing ℓ is better. However, there are practical limitations on the length, of course. We must use passwords that can be remembered and entered correctly. Experimental studies show that 12 digits seem to be the maximum one can expect anyone to enter correctly under stressful circumstances.

However, the number of possible passwords is not a measure of quality by itself. We have to worry about how they are chosen. Passwords that real people can remember are usually not uniformly distributed and are therefore often much easier to guess than what the password length might lead you to believe. One idea that helps a lot is to have people choose, not a password, but a *passphrase* and select the password letters from the passphrase. For instance, "I don't want to pass my exam" becomes "Id'twtpme". Studies of the behavior of actual users show that this can lead to passwords that are as hard to guess as random ones, but as easy to remember as "meaningful" passwords.

Even if passphrases are not used, you can still help people a lot to choose better passwords. Some systems test chosen passwords against known lists of "bad" passwords, and simply refuse to let users use them. This can be extended with checks on whether a new password has been used recently or is too similar to the previous one. Other systems try to measure quality of the password as it is typed in by the user and show the result on screen. The hope is that if users can see directly how well they are doing while choosing passwords, they will tend to make better choices.

Of course this creates a tradeoff, and a system that puts too many restrictions on the user might not only be harder to use, but even lead users to choosing *worse passwords* e.g., many users might end up choosing from a small set of simple passwords that fulfil all the criteria (e.g., *Password1!* is 10 characters long, includes upper/lower case letters, numbers and special symbols, but it is not very hard to guess).

A separate question in this context is: how easy is it for the adversary to verify his guess? Sometimes this can be quite easy, if the adversary has access in some way to the verifier's password storage (see below). Otherwise, the attacker has to try to log in and effectively use the targeted system to check the guess. This limits him in speed and may also limit the number of guesses, if for instance the account is frozen after 3 failed login attempts. Thus, limiting the number of login attempts may help security significantly. But one has to take care: if the attacker just wants to break into *some* account, he can just try once for every account, so in this case security depends on the total number of users, as well as the number of passwords and the max number of failed logins allowed. Another issue is that the attacker's goal might not be to get hold of a password, but to make the availability break down, which he has certainly succeeding in doing if he makes a large number of accounts freeze by just trying incorrect passwords. So whether limiting the number of failed login attempts is a good or a bad security mechanism depends on the assumed threat model. A better idea can be to have the system wait for some time after a failed login attempt before it allows the user to try again. And to make this time longer, the more failed attempts are made. For honest users, this will be a minor problem, but an attacker will be slowed down significantly.

10.6.2 Using and eavesdropping passwords

Stealing the password when it is used can take many forms. Watching over someone's shoulder while the PIN code is entered is a trivial but often very effective attack. A fancier variant is the scam where an ATM (Automatic Teller Machine) is equipped with a false front and a video camera, that records both the magnetic stripe of your credit card and your PIN code, thus enabling the crook to copy your card and start emptying your account minutes later. Finally, passwords sent in clear over a local area network are very easy indeed to detect and grab, and this has been done in practice on many occasions (for instance, hackers sitting at the local cafe intercepting other customers using the free wifi to login onto web services which do not make use of encrypted communication). Note also that looking over someone's shoulder can also be done electronically using so called spyware, where you get code to run on the victim's machine in a unnoticed way. The program will record passwords and send them to the attacker. We will have more to say about malicious software in general later. General precautions are hard to define because the threat models are so diverse, but encrypting network traffic definitely helps. In other cases, one can try to ensure that, in addition to stealing the password, the attacker must also get hold of something that is harder to get without the user noticing, this is the idea behind replacing the old DanKort by chip-cards – copying the information in the chip essentially requires to get hold of the DanKort and breaking open the chip.

10.6.3 Storing and stealing passwords, the user side

Stealing the password from the user's permanent storage can be easy if it is written down (either on a post-it on the computer monitor, or stored in plaintext on a hard-disk so that the user does not have to type it every time) but other methods can also be effective, such as trying to fool users into revealing their passwords under false pretenses. This is known as *social engineering*. In a study at University of Sydney, 336 students were sent an email asking them to send their password to a certain mail address. The mail claimed that this was necessary to validate the password database after a suspected attack. 138 of them returned a valid password. Many were suspicious and changed their passwords, but essentially no one reported the incident to the system administrator. A variant of this idea is the *phishing attack*, where you send a mail to the victim, claiming you are his bank, for instance. The mails asks the user to follow a link, leading to a fake web page that claims to be his bank's. Here the user is asked to type his user name and password, to "validate the account", for instance.

According to many people, including several hackers, social engineering is one of the most effective attacks on real systems, and systematically preventing it is very difficult. It seems the only mechanism that has some effect is to inform and train users, or to combine passwords with something else, such as hardware devices or biometrics (see later).

The problem seems to be that we are up against some basic human psychologic factors: it seems to be hard for us to assign value to passwords, perhaps because they are not a tangible objects we can hold in our hands. And therefore fooling people into giving them away seems much easier than, for instance, getting someone to give you the key to their house.

10.6.4 Storing and stealing passwords, the verifier side

Stealing passwords from the verifying party may also be possible. In some really bad cases systems store passwords in cleartext. UNIX (and several other systems) is somewhat better, in that the password file does not contain the passwords themselves, but instead a user record containing username u and $f(pw_u)$ where pw_u is the password of user u and f is a so-called one-way function., i.e., a sort of encryption where it is easy to compute $f(pw_u)$ from pw_u but hard to go in the opposite direction – this could e.g. be a hash-function like the ones discussed in Chapter 6. This way, the system can still verify a given password easily, but stealing the password file will not directly reveal any passwords. However, if the attacker has the password file and a password pw that he thinks is likely to have been chosen by *some* user, he can compute $f(pw)$ and check if this occurs in the file. This is a very relevant attack in UNIX because the UNIX password file is not too difficult to steal. (Over the last decade billions of password hashes have been leaked this way as a consequences of several breaches of security at large organizations).

Password crackers are programs that run this way: they use dictionaries with probable passwords and try to match them with entries in a given password file. Practical studies indicate that up to 25% of the passwords used in real life can be cracked this way, when people are not guided in any way or warned against bad choices of passwords.

There are three main countermeasures against these kind of attacks:

- *Educate the users choose hard to guess passwords:* encourage or even enforce better ways of choosing passwords, such as using the passphrase method above, or modifying the program that accepts new passwords such that it will reject "bad" passwords.

- *Slow the attacker down:* similarly to what was suggested for online login attempts, it is possible to make the life of the adversary harder by making sure that the function f is "slow enough" to slow down the number of attempts a password cracker can run per second, while at the same time "fast enough" for the system to be efficient enough. Some hash functions (such as e.g., bcrypt or PBKDF2), have been designed specifically for this purpose, and can get as input a parameter specifying how "slow" the function should be. Another very effective countermeasure is salting, which involves using some random number (called salt) when computing the hash of the function. (See more about salting Exercises 10.2 at the end of this chapter).
- *Removing the single point of failure:* an increasingly common architecture for storing passwords used e.g., by Facebook, is the following: passwords are hashed together with a secret key K which is stored on a different machine than the one storing the password hash. Therefore, the adversary must now get hold of both the hashed password and the key K to be able to verify their guesses. This of course also require the second server to be involved in every password verification attempt, but this seems to be a reasonable price to pay for the added security. More in detail, when the user u register its password pw, this is sent to the "hashing server" that computes $y = f(K, pw)$ and returns this value to the "authentication server" that stores the pair (u, y). When the user later contacts the authentication server, the authentication server sends the received password pw' and the hashed password y to the "hashing server" who checks whether $f(K, pw') = y$ or not.

10.6.5 Summary on Password Security

To recap, good advice on password security is as follows:

1. Make sure users have information and guidance that can help them to choose and remember good passwords, such as passphrases and on-line help.
2. Try to educate and inform users in order to prevent social engineering
3. Have the verifying system limit the ability to verify guessed passwords, by limiting the number of failed log-ons or by slowing down the check after failed attempts, and by using appropriate password hashing at the server side.
4. Make sure the verifying system stores passwords securely, using one-way encryption or hardware security.

10.6.6 A note on password managers

An increasingly popular way of dealing with creating and managing passwords are so called password managers. In a nutshell, a password manager is a piece of software that helps users generate "truly random" passwords and helps users by storing their passwords for them. Password managers solve the problem of users choosing easy to guess passwords, and users re-using the same passwords over multiple websites. Thus, they can greatly help against attackers who can get access to the hashes of the passwords stored at the verifier side. However, using password managers introduces new security issues: where and how are the password stored? Early *in browser* password managers would store the password in plaintext at the client side, making it very easy for an attacker that can hack into the user's device to recover the user passwords. More modern solutions store the passwords encrypted under a master password (using a method very similar to what is described in Section 10.9). Often, password managers allow users to access their password on multiple devices (computers, phones, etc.), and therefore store the encrypted passwords (sometimes referred to as one's *vault*) on some remote server or *in the cloud*. But, as someone famously said "there is no cloud, it's just someone else's computer". Therefore, if an attacker can now hack into the password storage server, the attacker can try to brute force the master password. As very often in security, it is not possible to make absolute statements such as "A is more secure than B". Instead, one has to look at a specific threat model, and then say "In thread model T_1, A is more secure than B" (which means that in thread model T_2, B might be more secure than A).

10.7 Hardware Security

10.7.1 Why use secure hardware?

Hardware units with some degree of physical security are very useful in many secure systems. The more obvious purpose is of course to prevent the adversary from getting hold of our secret keys. But it is very important to understand that this is not the only issue – it is also important to make sure that a certain secret parameter only exists in "one copy". An example: many home-banking systems store your private signature key on your hard disk, encrypted under a password. It may well be possible for an attacker to get a copy of the file with your encrypted key without you noticing this. Even though this may not reveal your key directly, he now has lots of time off-line on his own machine to try to find your password, which he can (at least in principle) do by trying all possibilities and testing them against the encrypted key.

A similar problem occurs with magnetic stripe cards that are very easy to copy, which means it is entirely possible to copy a card without the user noticing this, you just need to steal the card, copy it, and give the card back, posing as an honest citizen trying to help the poor user. This can be done in seconds.

If instead the key was inside a hardware unit that the attacker cannot easily break into, he may still be able to steal this hardware unit, but now he needs time (and money) to break into the device and runs a much greater risk that the theft is noticed and the account closed, before he has time to find the right password.

10.7.2 Tamper-evident hardware and two-factor authentication

Consequently, it is interesting to have hardware units that are *difficult* to break into, even if it is not *impossible* to do so. The unit may still be useful in improving security, as long as there is a significant cost, and if the attack will take some time and/or may leave some trace on the device showing it was attacked. Such hardware is often called *tamper-evident* hardware.

Chip-cards are examples of such units. They are basically cards with their own computer on board, complete with CPU, storage, operating system and various software put in by the manufacturer or distributor. They often come with built-in co-processors for encryption, in some cases even with a dedicated hardware implementation of RSA. The storage is physically protected such that access to using the keys on board should be only through the card's own CPU. This means you must know the PIN code that opens the card and follow the communication protocol. The best cards on the market have various protective layers put on top of the CPU and storage devices so that tampering with the card is hard without destroying the data on the card. Current state of the art is that breaking into a smart card is not impossible, but it requires a skilled and determined attacker, it takes non-negligible time, and requires specialized equipment. Thus chip-cards may be a very useful part of many systems.

One has to be careful, however, with the way one uses the processor(s) on the card. An example of this: Chipcards have to rely on external power-sources to operate, so an attacker may be able to continuously monitor how much power the card is consuming. The problem with this is that sometimes the card's power consumption depends on the instructions the CPU is currently handling, simply because some CPU commands require more power than others to execute. If the critical parts of code are not carefully done, this may reveal information on the keys used by the card. An example: the standard algorithm for doing an RSA decryption scans the secret exponent d bit by bit; it executes one set of operations if the bit is 0, another if the bit is 1. In a straightforward implementation, the operations needed for a 1-bit may consume a significantly different amount of power than those for a 0-bit, and so by plotting power consumption against time, the adversary will see a distinct pattern for every 1-bit, and so can read off the secret key directly!

In general, this shows that information can escape from hardware devices in unexpected ways, and one should not just consider the possibilities for output that the device was designed for.

A primary application for tamper-evident hardware is for so-called *two-factor authentication* where the idea is that we authenticate a user in two ways: first by a password and second by checking that he has a certain hardware unit in his possession. This is typically done by putting a secret key K inside the unit. This key is also held by the verifying party. The verifier issues a challenge c (a nonce) which the user forwards to the hardware unit, it then returns a response $R(K, c)$ that is a function of both K and c, typically it just encrypts c under K. The response can clearly be verified by the other party. The idea is of course that replay attacks will not work because the challenge will not be the same the next time, and therefore the hardware unit really needs to be involved in order for the authentication to be successful. The response $R(K, c)$ is often called a *one-time password*. In some cases, the challenge c is replaced by the current time, this frees the user from manually forwarding c to the device.

Lots of one-time password devices are on the market, as well as tools that are not technically one-time password devices and instead approximate this functionality, in a way which is more convenient for users. One well-known example is the plastic card from the Danish NemID system – which is essentially a one-time password devices where the codes have been pre-computed and printed on a piece of paper. Another example are "authenticator apps" that you can download on your smartphone (as used by e.g., Google, Facebook, etc.). In this case the secret key K will be transferred to your phone in a secure way during registration.

The big security advantage of two-factor authentication is of course that it is no longer possible to steal a password from the user and then freely impersonate him. However, it is important to realize that two-factor schemes can still be attacked, because the one-time password only depends on the user key and the nonce, and not on what the user actually wants to do once he is logged in. As a result one can do so called *real-time phising*: say the user wants to log into his bank. The adversary fools the user to go to his web page rather than that of the bank and log in there instead. At exactly the same time the adversary logs into the bank in the name of the user, forwards the nonce to the user and relays his response to the bank. He can now spend money in the user's name.

Defences against real-time phishing include: 1) have the user confirm the actions he wants via a different channel: some banks send you an SMS you must respond to, in order for the money to actually move. And 2) have the one-time password depend also on the transaction the user wants. This is harder, because in order for this to buy us something, the hardware device must receive information on the transaction and show some information to the user on what it is doing in a form that the user can understand. In the bank example, it would need to show at least the receiver of the money and the amount. Note that if the device only communicates digital codes that are not humanly readable, we have not achieved anything: the device could still be authenticating a transaction that originates from the adversary.

10.7.3 Tamper-Resistant hardware

Other types hardware units are much more difficult to break into. These are called *tamper resistant* or *tamper-proof*. There are devices from IBM, for instance the IBM 4758, that have been certified by the US administration to the highest possible level of tamper resistance. This means that it is estimated that not even a well funded organization with expert knowledge can break into the device. The 4758 is a plug-in card for PC's. Its protection mechanisms are alive at all times and will erase all internal keys if an attack is detected. It has it's own CPU, storage and battery back-up and can do all the standard cryptographic algorithms internally. Banks use these units to protect particularly sensitive and long-lived data, such as PIN codes for credit cards. Also CA's use this kind of technology to store their private keys used for issuing certificates. Currently no attacks are known to break into a 4758.

A final word of warning on hardware security: the old saying about the chain and the weakest link is also true here. Even the best hardware security will not help you if the system using the hardware can be fooled. An example: suppose I use a PC with an attached chip-card reader to sign a message, where my private key is on the card. This means that it will be very difficult for an attacker to get my key in cleartext. But if software running on the PC is not trustworthy, it may show me an innocent looking message on screen to make me click OK and type my PIN code, and then send a completely different message to the card, which will have to sign whatever is sent to it, once the PIN code has been entered.

10.8 Biometrics

Although today's technology usually protects access to hardware units by PIN codes or passwords, there are also other possibilities: the unit may try to recognize various physical characteristics of the person trying to get access. This is known as biometrics and can take a great number of different forms: the machine can scan your fingerprint, your eye, your face, or listen to your voice, to mention a few examples. From a logical point of view, all the solutions are similar, however: the physical characteristic measured is converted to a piece of digital data, which is then compared to an entry in a database created ahead of time.

This is more or less the same as we humans do when we recognize a face or a signature, except that the database is in our brain and we can use the very powerful pattern recognition capabilities that humans have. A computer has a much harder time, since even measuring the correct people will create certain variations in the result, for instance, hardly anyone writes a signature the same way every time. Therefore, the major problem in this area is always to do the conversion to digital and the comparison such that the system is tolerant enough to accept the good guys, but restrictive enough to reject the cheaters.

However, there are systems on the market today using some of the more "stable" characteristics, such as fingerprints or iris scans, that have quite an acceptable performance.

The big advantage of biometrics is of course that you don't need to remember the password, in a sense, you carry it with you.

There are also problems, however: depending on how biometrics is used, it can lead to privacy problems: whereas I can be anonymous towards a system using passwords, just by choosing a password and username that are unrelated to my physical identity, this is of course not possible if biometrics is used.

A common misunderstanding is that biometrics can replace cryptographic authentication. This is obviously false: biometrics can create a very reliable representation of your fingerprint, for instance, but sending this along with a message proves nothing because there is no connection between the "signature" and the content of the message – the message content can still be changed by anyone. What biometrics *can* do is to provide better access control for your private signature key. Finally, we have to mention the chain and the weakest link again: for instance, the database containing the samples to which measurement results are compared is clearly just as critical to security as the hardware unit doing the measurement. If its content can be falsified, then anyone can get in. Or, assuming the database is not stored in the same location as the unit doing the measurement, we clearly need authenticated communication between the two. If this is not done, an adversary could disconnect the real database and send to the measurement unit falsified messages claiming that the database verified the measurement successfully.

10.9 Preventing bypass of the system

A final issue we have to look at is the fact that all the techniques we have seen, passwords, hardware, and biometrics, are techniques to *identify* a user who wants access. If the identification checks, the system will grant the user access to whatever resource he wants. But of course, this is not useful, unless we make sure that the system cannot be bypassed, i.e., it must not be possible to get to the resource without asking the system.

If we have secure hardware available, and our resource is a key, this is not too difficult: just put the key inside the hardware and make sure it cannot be output from the device. We don't mean to say that ensuring this is trivial, but at least life is simpler when you know exactly how a key can leak, namely only if the hardware device explicitly outputs it.

If no secure hardware is available, life is much more difficult, nevertheless this is the situation if you are designing security on a standard PC. We look at a specific illustrative example here: suppose your private RSA key is to be access protected based on a password. Since we cannot store it with hardware security, the only option is to encrypt it, using the password as key. One problem with this is that a password is a character sequence and not a binary string of fixed length, like a 128-bit AES key. This can be solved by hashing the password using some standard hash function h and using 128 bits of the output as key. So for password pw and secret key sk, what is stored is $E_{h(pw)}(sk)$.

The question now is if one can get hold of sk without knowing the correct password? The problem here is that the number of possible passwords is often very small, typically much less than the 2^{128} possible AES keys. This means that unless we take precautions, an attacker can steal the file with the ciphertext and try all possible passwords. A solution is to make the function h be slow, i.e., rather than having h be a standard hash function such as SHA-1, we make it be many iterations of it, i.e.,

$$h(pw) = \text{SHA-1}(\text{SHA-1}(\ldots \text{SHA-1}(pw)...)),$$

say several thousand iterations (this is just an example to explain the main idea. As mentioned above, specific hash functions have been designed specifically for the purpose of hashing passwords). This will mean that the right user (who only has to compute h once) will be slowed down, but not too much, whereas the attacker will have a much harder time: he can try all possible AES keys, which is infeasible, or he can try all passwords, which is now much slower because h has to be computed for all possible passwords. Careful choice of the number of iterations can give quite reasonable security in practice of this type of system. Such functions, designed to be slow but not too slow, are sometimes called *moderately hard functions* and also have other applications known as *proof-of-work*, originally introduced for fighting spam, and nowadays being extensively used in popular cryptocurrencies such as Bitcoin.

10.10 Exercises

Exercise 10.1 Consider the following two systems for user authentication when logging into some computer system. For concreteness, let us assume we are implementing access control to a web site, so that the user is using some browser when logging in (at a very abstract level, this exercise describes two of the main methods that can be used to login using SSH, namely password-based or key-based).

1. The UNIX-like system: the host stores passwords in one-way encrypted form, that is, instead of password PW, $f(PW)$ is stored, where f is a one-way function, i.e., given PW it is easy to compute $f(PW)$, but given $f(PW)$ it is hard to compute PW. The user types username and password, which are sent to the host who checks this against the password file in the obvious way.
2. A digital signature based solution: the host stores a file containing public keys of registered users. At login, the host generates a random message R which is sent to the user. The user (or rather, his browser) signs R and sends the signature and user name back to host who the looks up the public key and verifies the signature.

Question 1: Consider a passive attacker who observes the communication of a user and then later tries to impersonate him. How does the security of the two systems compare if host and user communicate in the clear? What if an encrypted connection is used?

Question 2: Consider an attacker who breaks into the host and reads the file with information on users (password file or file with public keys). How does security of the systems compare if the attacker can only read (and not modify) this file? What happens if he can modify it?

Question 3: From a general point of view (including for instance also attacks that try to break into a user's PC), which system would you prefer (motivate your answer)?

Exercise 10.2 The one-way function f used in (some versions of) the UNIX system to "one-way encrypt" passwords before storing them in the password file is a modified version of DES encryption, where however there is no secret key that would allow easy computing backwards from $f(PW)$ to PW. One characteristic is that the function has been deliberately changed so that it is much slower than DES encryption. What could be the purpose of this?

In many cases the password file does not simply store $U, f(PW)$ for the password PW of user U, instead a technique known as "random salt" is used. That is, for each user U a random value R_U is chosen, and we store $U, R_U, f(PW||R_U)$.

Explain why the version with salt is more secure against attacks where the password file is known to the attacker.

Exercise 10.3 (Rainbow Tables) In this exercise, we consider the case where an attacker learns $f(pw)$ for a password pw and a hash function f, and wants to find pw. If there are N possible passwords, one can of course find pw by spending $O(N)$ time trying all possibilities.

But now suppose the attacker knows the function f ahead of time and is able to invest large resources in doing precomputation, so that once he gets $f(pw)$, he can find pw much faster than if he had to start from scratch.

One trivial way to do this is to make a sorted list with entries of form

$$(f(pw_1), pw_1), (f(pw_2), pw_2), \ldots$$

where pw_i runs through all possible passwords. With such a table one can just look up the target value $f(pw)$ and read out the desired value of pw directly. This uses almost no on-line computation, but one needs $O(N)$ memory which is usually not practical.

Here is an idea for a different method that uses less memory at the expense of more on-line time and can therefore lead to a more pactical attack:

We let g be a function that can take a hash value as input and output a string that might be a password. For instance, if f is SHA-256 and passwords are 7 characters, then g could simply output the first 56 bits (7 bytes) of the input. We define a function F by setting $F(pw) = g(f(pw))$. Note g is not introduced because we hope it will return the target password, there is no real chance that this will happen. We just want a way to construct from f a new function that maps into the password domain, since this turns out to be useful later.

Now, the pre-computation works as follows, using parameters t and n:

- For $i = 1$ to t: choose random password pw_i, and then compute $y_i = F^n(pw_i)$.
- Store a table T with entries (pw_i, y_i), sorted by the y_i-values.

Here, F^n means F iterated n times, $F^n(pw) = F(F(\ldots F(pw)\ldots))$.

Consider a matrix M with t rows and $n+1$ columns, where $M[i,j] = F^j(pw_i)$. We let j run from 0 to n and define $F^0(pw_i) = pw_i$. Note that this means that T contains the first and last column of M.

1. Describe an algorithm that uses T to find pw from $f(pw)$, under the assumption that all the y_i's are distinct, and that the pw we are looking for is found somewhere in the first n columns of M, i.e., for some i, j where $j \leq n$, $pw = M[i,j]$.

   Remember that we do not store M, only T.

Hint: Begin by computing y, defined as $y = g(f(pw)) = F(pw)$. Now, if indeed pw is in M, this means that for some i and j, we have $pw = F^j(pw_i)$. So it will be useful if you can find the right value of i and j. For this, note that $pw = F^j(pw_i)$ implies that $y = F(pw) = F^{j+1}(pw_i)$.

2. How many times does your algorithm need to compute F? Argue that your answer is correct.
3. What might go wrong if not all y_i's are distinct and (optionally) what could you do in the preprocessing to avoid this problem?

In general there is no guarantee that the right pw is found by the algorithm, since the target value may not be in the matrix M at all. But Martin Hellman, who had the basic idea, developed it further using several different tables and a smart choice of t and n to get an algorithm that uses memory and on-line time $O(N^{2/3})$ where N is the number of possible passwords.

Exercise 10.4 Consider the following system for providing digital signatures, for instance in a home banking system.

The private key of each user is stored in a chipcard held by the user. The card must be inserted in a card reader attached to the user's PC and activated by typing a PIN code. When a digital signature is to be generated, software on the PC computes a hash value on the message to be signed. This hash value is sent to the card and the card returns the signature to the PC.

Question 1: Some card readers come with their own keyboard and the PIN code must be typed on this keyboard. Other readers are without keyboard and the PIN is typed on the user's PC and transmitted to the reader. Compare the security of the two types. What must an attacker do to break the security in the two cases?

Question 2: Describe possible security problems that may exist if the software computing the hash value of the message is not reliable, for instance it is infected by a virus.

Question 3: Some systems require that the PIN code is typed for every message signed, i.e., the user must explicitly confirm that he actually wants to sign each message. Would such a requirement solve the security problems you identified in Question 2? Why or why not?

Question 4: Suppose private keys are instead stored in a hardware unit attached to a central server, and a user must authenticate himself to the hardware unit in order to have the unit use his privates key for signing messages. Compare the security of such a system to the one above. Note this solution is essentially the one that DanID uses for the Danish digital signature system from the fall of 2009. Here the authentication uses one-time passwords in the first version.

Exercise 10.5 Many homebanking systems store a private RSA key on the harddisk of the user's PC, encrypted under the user's password. Typically, the password is hashed and the hash value is truncated to obtain an AES key, for instance. The point of the hashing is that even if the password has different length than that of the key and uses some restricted character set, we can still generate approximately as many different keys as the number of different passwords.

A point that is often raised about such a system is that if the threat model includes an attacker that gains access to the disk of the PC, the security is no better than that of the password. This makes it essential to pick high quality passwords. Now answer the following: how long must the password be in order to have as many different passwords as there are AES keys if

- the password contains only digits (0-9)?
- the password contains digits and lower case letters?
- the password contains all characters one can directly type on a PC keyboard? (the answer here can only be approximate, since keyboards differ in the number of characters you can type).

Exercise 10.6 The IBM 4753 secure hardware crypto unit has a master key MK that is unique to each device and is always securely stored inside the device. This key is a 112 bit key for the so-called two-key triple DES algorithm. All keys in this exercise are of this type, but the details of two-key triple DES are unimportant here.

Of course, a 4753 has to work with lots of other keys besides MK. To avoid having to store all such keys inside the device, there is a mechanism for storing keys in external (insecure) storage, encrypted under MK.

However, there is an extra complication in this connection: in the IBM security architecture, one can restrict the usage of keys, for instance say that a particular key can only be used for encryption, only for authentication, only for decryption, etc. To each key is associated a so-called control vector CV, a bitstring that specifies what the associated key can be used for.

It is important that the 4753 cannot be fooled into using a key for the wrong purpose. Therefore, when a key K with control vector CV is stored externally, it is not enough to just store the pair $E_{MK}(K), CV$. Anyone could change CV and make the 4753 use the key according to the new control vector value. A natural solution would have been to include a MAC on the stored pair, but IBM did not want to use the extra space needed for this. Instead, they wanted a solution where the CV is used in the encryption and decryption process in such a way that if the CV is modified, this leads to a useless result when the key is decrypted inside the 4753. Here follows three proposals for what we could put in external storage:

1. $E_{MK}(K \oplus CV),\ CV$
2. $E_{MK \oplus CV}(K),\ CV$
3. $E_{MK}(K) \oplus CV,\ CV$

where $\oplus$ means bitwise xor, and we assume for simplicity that CV, MK and plain- and ciphertexts all have the same lengths.

Which solution(s) would you recommend for IBM to use and which would you recommend not to use?

Exercise 10.7 When a PIN code for a credit card is entered at a terminal, it is usually not verified at the terminal. Instead, the data on the card and the PIN code that was typed are sent in encrypted form to a central computer. The standard architecture is that this computer has on its harddisk a database containing entries of the form $(Cardnumber, EncryptedPINcode)$, where the PIN codes are all encrypted under a key that is known only to a secure hardware unit attached to the central computer. This hardware unit is able to decrypt internally the data sent from the terminal. It then reads the relevant record from the database, decrypts the PIN code and compares to the PIN that was typed by the user.

There are two standardized ways to do the encryption of the PIN code: ISO-0 format, which is simply $E_K(pin)$ where K is the key and pin is the PIN code. And ISO-1 format which is $E_K(pin \oplus cardno)$ where $cardno$ is the number identifying the card and account of the user that has PIN code pin (this is not well defined if the pin is shorter than the account number, you may assume one just repeats the pin enough times to get a string that is long enough).

Compare the two options in terms of attacks that might be possible by employees with access to the central computer.

Exercise 10.8 The NemID system is a Danish system for getting a public-secret key pair generated and obtain a certificate for your public key. What they do when you register is to send a paper letter to your registered address with a PIN code. This is then used when you log in the first time to get your certificate. What do you think about this procedure? Comment on the security.

Open the browser you normally use and try to investigate the details of the certificate you have (if it is there, this is not always the case). Also try to make make it show you the root certificates it has installed. This may be available in different places in different browsers. Your browser will trust any site with a certificate that can be verified using any of the root certificates on the list. What did you see? Are you surprised? Worried?

Exercise 10.9 Consider a password system controlling access to a multiuser system. Let L be the lifetime of a password, S the number of possible different passwords, N the number of users, and R the maximal login rate, i.e., the maximal number of login attempts for a single user that the system will accept in one time unit before an alarm is sounded. Concretely, this means for instance that if time is measured in minutes and $R = 10$ then if the system detects more than 10 failed login attempts from the same user within one minute, then this is assumed to be an attempted attack, and the system administrator is alerted.

Now, assuming passwords are chosen at random, and that we only consider attacks where the adversary tries to log in as a particular user using some guess at the password, derive an expression in terms of L, S and R for the probability that the adversary finds the password of one particular user during one life cycle of a password, without the attack being detected.

Assume for simplicity that all users change passwords simultaneously. Find an expression in terms of L, S, N and R for the probability that the adversary could find the password of SOME user, during one life cycle, without the attack being detected.

Exercise 10.10 (Nem ID) When you get a credit card from some Danish banks, they send you the PIN code in a separate paper mail. In the envelope is a little paper card that is supposed to help you remember your PIN. It is essentially a matrix with 5 rows and 8 columns. The instructions tell you to fill the 4 digits in your PIN into positions in the matrix, chosen in some pattern you can remember. Then you fill in the rest of the entries with random digits.

The hope is that an attacker getting hold of this card will not learn anything about your PIN.

Consider an attacker that looks at the card and chooses an ordered set of 4 entries. He knows, of course, that if this happened to be the entries chosen by the user, he is looking at the correct PIN. In other words, every ordered choice of 4 entries corresponds to a PIN code that is possible given this particular card.

A necessary condition for security is, of course, that with high probability, all PIN codes remain possible, given the filled-in card. To see if this is the case, consider the following: since there are 40 positions on a card and 10 different digits, we expect to see, on average, 4 instances of each digit, i.e., a typical filled-in card would have about 4 zeros, for instance. But of course there might be fewer than 4 zeros.

1. Show that the probability that a fixed digit, say zero, does *not* occur on a card with randomly chosen digits, is approximately 0.015. If this event occurs, how many PIN codes can an attacker exclude after seeing the card?
2. A closer look will reveal that in fact some special PIN codes have an even larger probability of being excluded than what the above analysis might suggest: Calculate numerically the probability that a card with random digits has exactly 1 zero. If this happens, which and how many PIN codes would be excluded? Optional: Generalise your result.
3. Do you see any issues with security that are not covered by the mathematical model?
4. What is your conclusion on security of this device – is it a good idea to send such cards to users?

Exercise 10.11 (Software Wallet) Use your solution from Exercise 6.10 to create a software wallet for an RSA secret key. It should have these functions.

1. `Generate(filename string, password string) string` which generates a public key and secret key and places the secret key on disk in the file with filename in `filename` encrypted under the password in `password`. The function returns the public key.
2. `Sign(filename string, password string, msg []byte) Signature` which if the filename and password match that of `Generate` will sign `msg` and return the signature. You pick what the type `Signature` is.

Your solution should:

1. Make measures that make it costly for an adversary which gets its hands on the keyfile to bruteforce the password.
2. Describe clearly what measure have been taken.
3. Explain why the system was designed the way it was and, in particular, argue why the system achieves the desired security properties.
4. Test the system and describe how it was tested.
5. Describe how your TA can run the system and how to run the test.

11

State Machine Replication (DRAFT)

Contents

In this chapter we cover state machine replication. When a service in a distributed system is important its continued operation can be secured by replicating it. This could for instance be an important database or the file system, which we would like to keep working even if a single physical machine crashes.

Replication just means that you run the service on several computers and keep them consistent. To give input to the system you give the input to at least one correct server that will then send it to the other servers. Then they each run their copy of the service on the given input. That way they all end up with an up-to-date copy of the service. If for instance you want to make a change to a file, the file is changed on all the replicas. It is clear that some mechanism must be in place which ensure that updates are applied in the same order at all replicas, we discuss this issue later. If one of the replicas crashes, you can use any of the other ones. If you want to tolerate malicious behaviour you will need at least three servers and in that case trust that at most one will be corrupted. In that case, to receive an output from the system you need to receive the output from all three servers and if there is disagreement adopt the one that was sent by at least two servers. If for instance you want to read your file, then you get all three copies and use the one that occur at least twice.

11.1 State Machines

For the above to work it is important that if you have to identical copies of the service and you implement the same change on both, then they end up being identical again. That might sound trivial, but it for instance excludes services using randomness. We say that the service need to be deterministic.

We abstract the service to be replicated by a so-called state machine M.

Definition 11.1 (State Machine) A state machine M which consists of:

- A set States.
- A start state $\mathsf{State}_0 \in \mathsf{States}$
- A set Inputs.
- A set $\mathsf{Outputs}$.
- A transition function $T : \mathsf{States} \times \mathsf{Inputs} \to \mathsf{States} \times \mathsf{Outputs}$.

A state machine starts in State_0. When it is in state State_i and receives input x then it computes $(\mathsf{State}_{i+1}, y) = T(\mathsf{State}_i, x)$, changes state to State_{i+1} and outputs y.

$\triangle$

11.2 Replicated State Machines

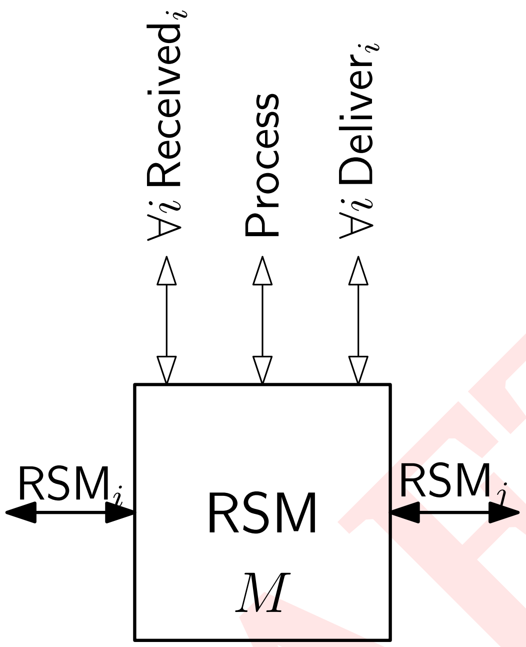
A replicated state machine is a protocol for n servers, which makes them behave as if they are running one single state machine M. We now formalise in more detail what a *replicated* state machine is supposed to do. We do this via an ideal functionality. This is an IA with the intended behaviour, as when we used a channel like LAC to specify the intended behaviour of a lossy authenticated channel. If fact, an ideal functionality is *just* a channel in the sense of Chapter 3. But since RSM has a much more complicated behaviour than just moving messages around, it can be a bit confusing to call it a channel. We therefore sometimes use the term ideal functionality.

Definition 11.2 (Replicated State Machine) Let M be a state machine. A replicated state machine running M is specified via an ideal functionality RSM_M for n servers $\mathsf{S}_1, \ldots, \mathsf{S}_n$. The syntax is as follows:

- There is a protocol port RSM_i for receiving inputs from server S_i and giving outputs to S_i.
- There is a special port $\textsc{Received}_i$ for reporting what messages have been input to the ideal functionality by S_i.

- There is a special port PROCESS for specifying which message to process next.
- There is a special port DELIVER_i for instructing the ideal functionality to deliver the next message to S_i.

The ideal functionality is depicted here:


$\forall i$ Received_i
Process
$\forall i$ Deliver_i
RSM_i
RSM
M
RSM_j

. The ideal functionality runs as follows:

- Upon initialiation, let $\mathsf{State} = \mathsf{State}_0$. For each RSM_i, let Q_i be the empty queue. The queue Q_i is the outputs for server S_i which have not been delivered yet. Initialise $\mathsf{UnProcessed}$ to be the empty set.
- On input x on RSM_i, output x on RECEIVED_i and add x to $\mathsf{UnProcessed}$.
- On input x on PROCESS, where $x \in \mathsf{UnProcessed}$, let $(\mathsf{State}', y) = T(\mathsf{State}, x)$ and update $\mathsf{State} = \mathsf{State}'$. Then add y into all the queues Q_i. Then remove x from $\mathsf{UnProcessed}$.
- On an input on DELIVER_i where Q_i is not empty, remove the front element y from Q_i and output y on RSM_i.

$\triangle$

One can formulate a liveness property which basically says that if x is added to $\mathsf{UnProcessed}$, then it will eventually be processed and that all outputs will eventually be delivered. We will not do this in more formal detail.

An important safety property which is evident from how the ideal functionality is formulated, is that the outputs that the parties see is always the result of running from the initial state on some sequence of inputs that is the same for all parties. Notice that the parties using RSM_M might not have seen the same number of these outputs.

11.3 Totally-Ordered Broadcast

The main building block to implement state-machine replication is totally-ordered broadcast as defined here.

Definition 11.3 (Totally-Ordered Broadcast) We specify the behaviour of totally-ordered broadcast via an ideal functionality TOB for n parties $\mathsf{P}_1, \ldots, \mathsf{P}_n$. The syntax is as follows:

- There is a protocol port TOB_i for receiving inputs from server P_i and giving outputs to P_i.
- There is a special port $\textsc{Received}_i$ for reporting what messages have been input to the ideal functionality by S_i.
- There is a special port $\textsc{Queue}$ for specifying which message to queue next.
- There is a special port $\textsc{Deliver}_i$ for instructing the ideal functionality to deliver the next message to P_i.

The ideal functionality runs as follows:

- On initialisation, let $\mathsf{UnQueued} = \emptyset$. For each TOB_i, let Q_i be the empty queue. The queue Q_i is the outputs for P_i which have not been delivered yet.
- On input x on $\textsc{in}_i$, output x on $\textsc{Received}_i$ and add x to $\mathsf{UnQueued}$.
- On input x on $\textsc{Queue}$, where $x \in \mathsf{UnQueued}$, enter x into all the queues Q_i, and then remove x from $\mathsf{UnQueued}$.
- On an input on $\textsc{Deliver}_i$ where Q_i is not empty, remove the front element x from Q_i and output x on TOB_i.

$\triangle$

We now argue that if one can implement TOB, then it is very easy to implement RSM_M. So in some sense, implementing TOB is the hard part of implementing replicated state machines.

To implement RSM_M given TOB we simply broadcast all inputs to all servers, which then run the machine M on the agreed sequence – therefore, if everyone runs the same machine on the same sequence of inputs, all the servers will end up in the same state. The protocol is given in Fig. **??**

We can then focus our attention on how to implement TOB. We already saw an implementation in Chapter 4, but there we assumed that all processes are correct. Here we will make more realistic assumptions, for instance that a significant fraction of the processes are Byzantine.

11.4 Crypto Currencies I

Before we discuss implementations of totally-ordered broadcast, we discuss a popular use of state machine replication known as cryptocurrencies. This basically just means that we replicate a machine which keeps track of the holding of some accounts. The idea of the Bitcoin project was to do it in a peer-to-peer network, where everybody can run a replica. That way one would have a peer-to-peer

The protocol has n servers S_i and uses TOB.

- Each S_i has a port $\mathrm{RSM}_M.\mathrm{RSM}_i$ for receiving inputs and giving outputs to P_i.
- Each S_i has a port connected to $\mathrm{TOB}.\mathrm{TOB}_i$ so it can give TOB inputs and outputs.

The protocol runs as follows:

- S_i: On initialization, initialise TOB and let $\mathsf{State} = \mathsf{State}_0$.
- S_i: On input x on $\mathrm{RSM}_M.\mathrm{RSM}_i$ input x on $\mathrm{TOB}.\mathrm{TOB}_i$.
- S_i: On output x on $\mathrm{TOB}.\textsc{out}_i$, let $(\mathsf{State}', y) = T(\mathsf{State}, x)$, update $\mathsf{State} = \mathsf{State}'$ and output y on RSM_i.

Figure 11.1 Replicated State Machine for $M = (\mathsf{States}, \mathsf{State}_0, \mathsf{Inputs}, \mathsf{Outputs})$ from Totally-Ordered Broadcast

currency without a single point of attack and which was not controlled by any nation state. If you solved Exercise 4.6, then you already saw a toy example of doing this in a peer-to-peer network.

Anyone can potentially send a command to move money between accounts. One way to prevent stealing of money is to let accounts are named by a verification key vk. The account owner has the corresponding signing key sk. To spend from the account, the transaction needs to be signed by the corresponding sk. This is to ensure that only the account holder can spend the holdings on the account. If you solved Exercise 6.13, then you already saw an example of this.

It is important to use totally-ordered broadcast to be able to agree on the order in which transactions are made: if there is an attempt to transfer more money from an account than it holds, the latest withdrawal has to be cancelled. To be able to do that, all parties need to agree which transaction was the last one. If some replicates cancel one transaction and other replicas cancel the other transaction, they will no longer hold identical copies of what is on each account.

In a bit more detail, the internal state of the machine is a list of pairs $(\mathsf{vk}, h_{\mathsf{vk}})$, where vk is the verification key for a signature scheme and $h_{\mathsf{vk}} \in \mathbb{R}_0$ is the holdings of the account. The holdings of the initial accounts is application specific. We will just assume that some accounts $(\mathsf{vk}, h_{\mathsf{vk}})$ are build into the machine when it is created. Typically the initial accounts will belong to the people that built the system and the ones that invested in the system.

On input $(\textsc{Transfer}, a, \mathsf{vk}_S, \mathsf{vk}_R, \sigma)$, where $a \geq 0$ and

$$\mathsf{Ver}_{\mathsf{vk}_S}(\sigma, (\textsc{Transfer}, a, \mathsf{vk}_R)) = \top$$

the machine will do the following: If $a < h_{\mathsf{vk}_s}$, then ignore the command. Otherwise, proceed as follows: If $(\mathsf{vk}_R, h_{\mathsf{vk}_R})$ exists, then retrieve it. Otherwise initialise it to $(\mathsf{vk}_R, h_{\mathsf{vk}_R} = 0)$. Then update $(\mathsf{vk}_S, h_{\mathsf{vk}_S})$ to $(\mathsf{vk}_S, h_{\mathsf{vk}_S} - a)$ and update $(\mathsf{vk}_R, h_{\mathsf{vk}_R})$ to $(\mathsf{vk}_R, h_{\mathsf{vk}_R} + a)$.

A cryptocurrency as described above is a layer which can in principle be put on top of any totally-ordered broadcast. How the standings on the accounts attain value is an economic problem outside the scope of this book. One way could be that there is a law which guarantees that the numbers can be exchanged for real money, as with your bank account. In the case of Bitcoin, the value simply emerged. People started to exchange Bitcoins for real money, and once this happened an expectation arose that the holds of Bitcoins could sell them for money in the future and that the price might go up. Then they became a speculation object and the price sky rocketed.

11.4.1 Smart Contracts

Some crypto currencies also have a notion of smart contracts. Treating the topic of smart contracts fairly belongs to a programming language course and is outside the scope of this book. We will, however, discuss it briefly.

A smart contract is simply a program P that can be "uploaded" to the replicated machine. Programs have to be associated to an account, so an upload would look something like

$$(\textsc{Upload}, P, \mathsf{vk}, \sigma)\ .$$

If

$$\mathsf{Ver}_{\mathsf{vk}}(\sigma, (\textsc{Upload}, P)) = \top\ ,$$

then P is added to replicated machine. Based on the state of the machine, the program P could make outputs of the form

$$(\textsc{Transfer}, a, \mathsf{vk}_R)$$

in response to which a is transferred from account vk to vk_R if there are sufficient funds. It can also make an output of the form $(\textsc{Signal}, s)$ in which case

$$(\textsc{Signal}, s)$$

is added to the state of the replicated machine such that the signal is visible by other smart contracts.

Technically, uploading a program would simply mean that you broadcast

$$(\textsc{Upload}, P, \mathsf{vk}, \sigma)\ .$$

Then let State be the list of all values that were sent on the totally-ordered broadcast channel at a given point. The program P is simply a function that is applied to State. Whenever State grows, the program P is applies again to see what it does in this new state. We could use $P(\mathsf{State}) = \bot$ to signal that nothing new happened. If $P(\mathsf{State}) = c$ for $c \neq \bot$, then it means that the smart program output c at this particular point. Here c could either be a transfer or a signal. In both cases c is added to State so they can be seen by other smart contracts.

Notice that each smart contract in principle is its own agent: it receives, process es,and send signals. The smart contract layer therefore in principle becomes its own emulated distributed system of agents on top of the distributed system of machine. However, the smart contract live in a shared memory environment. So the collection of smart contract lives more in the realm of parallel programming than in the realm of distributed system. The totally-ordered broadcast is basically just a glorified event queue and the smart contracts are consumers and producers of event. This is a common programming pattern, for instance in graphical user interface programming.

When you design a massively distributed software system like a smart contract layer, it is important to build in the possibility of parallelism. Early cryptocurrencies with smart contract like Ethereum designed their smart contract layer such that it is very hard to parallelise. Smart contracts can make instantaneous changes to the state of other smart contracts, and all contracts can see all signals. This ensures that most transaction are not concurrent and hence have to be executed in the same order everywhere. This is the kind of system we described above, where each P would get the entire State as input. Modern cryptocurrencies make much better designs where scopes are used to ensure that many operations can be run in parallel. One way to do it is to have each smart contract subscribe to only some messages, it could for instance give a list of other smart contracts which it want to be able to see. Each replica would then only input to P the transactions and signals to and from the smart contracts that P subscribed to. Consider then two signals which are not both visible to the same smart contract. Then it does not matter in which order they appear in State at different replicas. Put briefly, it is no longer necessary with totally-ordered broadcast. It is sufficient with causally-ordered broadcast, where two signals are considered causally related if they are subscribed to by the same contract. Independent contracts can also be executed in parallel locally, as they do not depend on each others' outputs. This allows to use multiple cores on a single replica to evaluate the contracts faster.

11.5 Synchronous Implementation of Totally-Ordered Broadcast

We first look at a rather trivial synchronous implementation of TOB. All parties will simply broadcast their messages. This ensure that all parties eventually see all the same messages. They might, however, see them in different order. To fix this we will elect a leader who gets to decide what the order should be.

In a bit more detail, the basic blue-print of the protocol is as follows:

1. When a new command x arrives at S_i, use unscheduled consensus broadcast (see Section **??**) to broadcast x to all servers. Recall that in unscheduled broadcast a party can broadcast when it wants to and then the message eventually arrive at all parties.
2. All servers S_i keeps a set $\mathsf{UnQueued}_i$ of all messages received above. These might have been received in different order.
3. All servers S_i also keeps a set Queued_i. They will be moving messages from $\mathsf{UnQueued}_i$ to Queued_i. When they do so they make sure to enter messages into Queued_i in the same order.
4. There is a designated leader L, often called a $\mathsf{sequencer}$, who will help the servers

keep the same order when they put messages into Queued_i. If L is corrupted it might harm liveness but not safety, specifically, if message are put into Queued_i then all servers do it in the same order. However, it might happen that no messages are put into Queued_i. Liveness is then guaranteed by leading the leader role go on round robin.

They way that the leader orders messages is that it broadcast a so-called block, which is essentially just an ordered list of messages that have still not been processed. Everybody outputs them in the ordered that the leader has chosen. It is important that all parties get the next block or not. Therefore we use scheduled broadcast, where everybody is guaranteed to get an output at the same time.[1] The reason we can use scheduled broadcast is that the leader is supposed to send the next block at a well-defined time.

The protocol will consists of two logically different parts. The first one is the flooding system which uses unscheduled consensus broadcast UCB for flooding all inputs. Since this part is very similar to the one for the synchronous system, we will leave it out of the protocol description in Fig. 11.3. The second part of the system creates an order on the unqueued elements. In addition to the flooding system it use a scheduled consensus broadcast SCB (see Section **??**). Remember that this just means that if some party is scheduled to broadcast, then all correct parties receive some message, even if the scheduled party is corrupted and tries to not send a message. We assume that SCB has the property that if all correct processes start running a scheduled broadcast in the same round, then they all terminate that broadcast in the same round. The protocol is given in Fig. 11.2

11.5.1 Analysis

We now argue that the protocol implements TOB. It is clear that the parties will agree, as the output U is the output of a consensus broadcast and they all order the outputs in U using the same deterministic rule. As for liveness, assume that x is input to a correct P_i. Then it will be input to the flooding system and will eventually arrive at all correct P_i. Therefore, in the first epoch where the leader P_i is correct, x will be put in U_i. Therefore x will be output by all P_i at the end of the epoch.

Note that a corrupted leader can add to U element x that were not broadcast. This is not a problem. A corrupted leader could get the same effect by first broadcasting x and then have it correctly be added to U. So we do not consider that an attack.

11.6 Asynchronous Implementation of Totally-Ordered Broadcast

We now look at a fully asynchronous system. Again the system consists of two different parts. The first one is the flooding system, which works using the following rules.

[1] Recall that in unscheduled broadcast the honest party only *eventually* gets the same output.

A protocol for n parties $\mathsf{P}_1, \dots, \mathsf{P}_n$. It uses

- An unscheduled consensus broadcast UCB (see Section **??**).
- A scheduled consensus broadcast, where all correct parties terminate in the same round SCB (see Section **??**).

The protocol consists of two parts. The first part works using the following activation rules.

- P_i: On input x on TOB.TOB$_i$ input x on UCB.UCB$_i$.
- P_i: On output x on UCB.UCB$_i$ add x to $\mathsf{UnQueued}_i$.

The second part works using the following activation rules:

1. Initially, let $\mathsf{UnQueued}_i$ and Queued_i be empty sets. Let $\mathsf{epoch} = 1$. The value epoch indicates who is the leader in a given epoch. The leader in epoch epoch is the P_i with $i = \mathsf{epoch} \pmod n$.
2. Do the following:
   - Leader P_i: Let $U_i = \mathsf{UnQueued}_i \setminus \mathsf{Queued}_i$, and input U_i on SCB.IN$_i$.
   - $\mathsf{P}_{j \neq i}$: Input P_i on SCB.IN$_j$ to indicate that P_i is broadcasting.
3. When the so-called block U is received on SCB.OUT$_j$, do the following: remove U from $\mathsf{UnQueued}_i$, add U to Queued_i, and output on TOB.OUT$_j$ the elements in U in some deterministic order, say lexicographically. Let $\mathsf{epoch} = \mathsf{epoch} + 1$. Go to Step 2

Figure 11.2 Synchronous Totally-Ordered Broadcast

- The protocol uses an asynchronous Byzantine agreement ABA (see Section **??**).
- Initially let $\mathsf{epoch} = 1$ and $\mathsf{Delivered}_i = \emptyset$ and $\mathsf{Received}_i = \emptyset$. Also for each $e \geq 1$ let

  $$\mathsf{Core}_i^e = \mathsf{CoreCandidates}_i^e = \mathsf{SeenByManyHonest}_i^e = \mathsf{HasSeen}_i^e = \emptyset \ .$$

  The use of the variables are as follows:
  - $\mathsf{HasSeen}_i^e$: This is the set of parties from which P_i saw a block in epoch e.
  - $\mathsf{SeenByManyHonest}_i^e$: This is the set of parties P_j for which P_i knows that at least $n-t$ parties claim to have seen the block from P_j.
  - $\mathsf{CoreCandidates}_i^e$: At a sufficiently late point in the protocol $\mathsf{CoreCandidates}_i^e$ will be set to be the union of the values $\mathsf{HasSeen}_j^e$ from $n-t$ other parties P_j. The logic of the protocol will ensure that in $\mathsf{CoreCandidates}_i^e$ there will be at lot of parties that will have their block seen by all honest parties.
- Each P_i runs the below activation rules. All values are sent via UCB.
  1. As the first thing in the new epoch, let $U_i^{\mathsf{epoch}} = \mathsf{Received}_i \setminus \mathsf{Delivered}_i$ and send $(\textsc{Block}, \mathsf{epoch}, U_i^{\mathsf{epoch}})$.
  2. When having received $(\textsc{Block}, \mathsf{epoch}, U_j^{\mathsf{epoch}})$ from P_j, where $U_j^{\mathsf{epoch}} \subseteq \mathsf{Received}_i$, add P_j to $\mathsf{HasSeen}_i^{\mathsf{epoch}}$, store U_j^{epoch}, and send $(\textsc{SawBlockFrom}, \mathsf{epoch}, \mathsf{P}_j)$ with $(\textsc{SawBlockFrom}, \mathsf{epoch})$ being the message identifier.
  3. When having received $(\textsc{SawBlockFrom}, \mathsf{epoch}, \mathsf{P}_k)$ from $n-t$ distinct parties add P_k to $\mathsf{SeenByManyHonest}_i^{\mathsf{epoch}}$.
  4. When it first happens that $|\mathsf{SeenByManyHonest}_i^{\mathsf{epoch}}| = n-t$, send the set $\mathsf{HasSeen}_i^{\mathsf{epoch}}$ to all parties.
  5. We say that $\mathsf{HasSeen}_j^{\mathsf{epoch}}$ has arrived at P_i when the value was received and for each $\mathsf{P}_k \in \mathsf{HasSeen}_j^{\mathsf{epoch}}$ the value $(\textsc{Block}, \mathsf{epoch}, U_k^{\mathsf{epoch}})$ was received. When $\mathsf{HasSeen}_j^{\mathsf{epoch}}$ has arrived from $n-t$ parties P_j, let

     $$\mathsf{CoreCandidates}_i^{\mathsf{epoch}} = \cup_{\mathsf{P}_j} \mathsf{HasSeen}_j^{\mathsf{epoch}} \ .$$

  6. When $\mathsf{CoreCandidates}_i^{\mathsf{epoch}} \neq \emptyset$, then for each P_k run a $\mathrm{ABA}_k^{\mathsf{epoch}}$ with input 1 if $\mathsf{P}_k \in \mathsf{CoreCandidates}_i^{\mathsf{epoch}}$ and input 0 otherwise. Here (epoch, k) is the identifier.
  7. When all $\mathrm{ABA}_1^{\mathsf{epoch}}, \ldots, \mathrm{ABA}_n^{\mathsf{epoch}}$ terminated, then for each P_k let v_k^{epoch} be the output of $\mathrm{ABA}_k^{\mathsf{epoch}}$. Let $\mathsf{Core}_i^{\mathsf{epoch}}$ be the set of P_j for which $v_j^{\mathsf{epoch}} = 1$, i.e.,

     $$\mathsf{Core}_i^{\mathsf{epoch}} = \{\mathsf{P}_j \mid v_j^{\mathsf{epoch}} = 1\} \ .$$

  8. When $\mathsf{Core}_i^{\mathsf{epoch}} \neq \emptyset$ and $(\textsc{Block}, \mathsf{epoch}, U_j^{\mathsf{epoch}})$ arrived for all $\mathsf{P}_j \in \mathsf{Core}_i^{\mathsf{epoch}}$, then let

     $$V_i^{\mathsf{epoch}} = \cup_{\mathsf{P}_j \in \mathsf{Core}_i^{\mathsf{epoch}}} U_j^{\mathsf{epoch}} \ ,$$

     and

     $$\mathsf{Delivered}_i = \mathsf{Delivered}_i \cup V_i^{\mathsf{epoch}} \ ,$$

     and output the elements of V_i^{epoch} in some deterministic order. Then let $\mathsf{epoch} = \mathsf{epoch} + 1$ and repeat the above activation rules for the new epoch.

Figure 11.3 Asynchronous Totally-Ordered Broadcast

- P_i: On input x on TOB.TOB$_i$ input x on UCB.UCB$_i$.
- P_i: On output x on UCB.UCB$_i$ add x to $\mathsf{Received}_i$.

The second part of the system is very different from the synchronous version. The reason is that we cannot risk waiting for any particular leader, as we might end up waiting forever: recall that in the asynchronous model we cannot tell the difference between a slow message and a message that was never sent. Furthermore, we do not have clocks, so we cannot use timeouts to interrupt an infinite wait. We will therefore in each epoch have all parties propose the next block and then make sure to wait until some honest parties had their block distributed to at least $t+1$ other honest parties. We say that such a block was *seen by many honest*. Then each honest party collects from $n-t$ parties all the blocks that these parties have seen. This means that all blocks which were seen by many honest parties will be collected by all honest parties. So now some blocks were seen by all honest parties. The final block will be the union of these blocks. To detect which block were seen by all honest parties and to ensure agreement on this, we will do some Byzantine agreements. The protocol is given in Fig. 11.3.

11.6.1 Analysis

We now argue that the protocol implements TOB when there are at most t Byzantine errors and $n \geq 3t+1$.

Agreement

We first see that the parties will agree on the outputs. It is clear that they agree on the set Core as it is uniquely defined via outputs of Byzantine agreements. They also agree on the sets U_j^{epoch} as these sets were sent via UCB and only one set U_i^{epoch} can be sent per session identifier. Hence they also agree on $V_i^{\mathsf{epoch}} = \cup_{\mathsf{P}_j \in \mathsf{Core}} U_j^{\mathsf{epoch}}$.

Liveness

Now let us check that if all messages are eventually delivered, then the system in fact will eventually deliver at all correct processes all the inputs input to a correct process. We first argue three simple properties, called Flooding, Progress and Large Core. We then put them together to argue the Delivery property.

Flooding

First of all, it is clear that if x is given to the flooding system at a correct process, then x will eventually enter all the sets $\mathsf{Received}_i$ of correct P_i.

Progress

We now argue that all correct processes eventually reach all epochs. Assume inductively that all processes eventually reach epoch epoch. We argue that then they all also eventually reach epoch $\mathsf{epoch}+1$. By assumption, all correct P_i will reach Rule 1 in epoch epoch and therefore send $(\textsc{Block}, \mathsf{epoch}, U_j^{\mathsf{epoch}})$. This message therefore eventually reaches all $n-t$ correct parties whom will send

$(\textsc{SawBlockFrom}, \mathsf{epoch}, \mathsf{P}_j)$. These $n-t$ messages eventually reach all $n-t$ correct P_i who will add P_j to $\mathsf{SeenByManyHonest}_i^{\mathsf{epoch}}$. Hence $\mathsf{SeenByManyHonest}_i^{\mathsf{epoch}}$ will eventually get to size $n-t$. So eventually all correct P_i send $\mathsf{HasSeen}_i^{\mathsf{epoch}}$ to all parties. This value will arrive at all correct processes P_j (as defined in Rule 5), as $\mathsf{P}_k \in \mathsf{HasSeen}_i^{\mathsf{epoch}}$ means that P_i saw $(\textsc{Block}, \mathsf{epoch}, U_k^{\mathsf{epoch}})$ and hence this value will eventually arrive at P_j. At this point $\mathsf{CoreCandidates}_i^{\mathsf{epoch}}$ is set to a non-empty set and Rule 6) is activated. Hence all correct processes start to run all the Byzantine agreements, which will therefore all eventually terminate, at which point Rule 7) is activated. We argue below that this sets $\mathsf{Core}_i^{\mathsf{epoch}}$ to a non-empty set, so for now we will just assume this fact. This triggers Rule 8). Now note that if $\mathsf{P}_j \in \mathsf{Core}_i^{\mathsf{epoch}}$, then $\mathrm{ABA}_j^{\mathsf{epoch}}$ output 1. This means that at least one correct P_k gave input 1 to $\mathrm{ABA}_j^{\mathsf{epoch}}$ meaning that $\mathsf{P}_j \in \mathsf{CoreCandidates}_k^{\mathsf{epoch}}$, which can be seen to imply that $(\textsc{Block}, \mathsf{epoch}, U_j^{\mathsf{epoch}})$ arrived at P_k earlier. Therefore $(\textsc{Block}, \mathsf{epoch}, U_j^{\mathsf{epoch}})$ will eventually arrive at P_i too. This holds for all $\mathsf{P}_j \in \mathsf{Core}_i^{\mathsf{epoch}}$, so Rule 8) will eventually be activated. At this point P_i proceeds to the next epoch as desired.

Large Core

We now argue that $|\mathsf{Core}_i^{\mathsf{epoch}}| \geq n-t$. Notice that no correct P_i sends $\mathsf{HasSeen}_i^{\mathsf{epoch}}$ until $\mathsf{SeenByManyHonest}_i^{\mathsf{epoch}}$ has size $n-t$. If $\mathsf{P}_k \in \mathsf{SeenByManyHonest}_i^{\mathsf{epoch}}$ then $n-t$ processes has reported to have seen U_k^{epoch}. Hence $n-2t$ correct processes has reported to have seen U_k^{epoch}. Hence when $\mathsf{P}_k \in \mathsf{SeenByManyHonest}_i^{\mathsf{epoch}}$ for a correct P_i then P_k is in the set $\mathsf{HasSeen}_j^{\mathsf{epoch}}$ for at least $t+1$ honest P_j. Let D_h be the set of P_j for which $\mathsf{P}_k \in \mathsf{HasSeen}_j^{\mathsf{epoch}}$. Putting this together we see that no correct P_i sends $\mathsf{HasSeen}_i^{\mathsf{epoch}}$ until it holds for at least $n-t$ processes P_k that $|D_k| \geq t+1$. Since all correct P_i collect $n-t$ of the sets $\mathsf{HasSeen}_j^{\mathsf{epoch}}$ it follows that each correct P_i will receive $\mathsf{HasSeen}_j^{\mathsf{epoch}}$ for at least one $\mathsf{P}_j \in D_k$. Hence P_k ends up in $\mathsf{CoreCandidates}_i^{\mathsf{epoch}}$ for all honest P_i. Therefore all correct P_i will input 1 to $\mathrm{ABA}_k^{\mathsf{epoch}}$ and therefore P_k ends if up in $\mathsf{Core}_i^{\mathsf{epoch}}$ at all correct P_i. Therefore $\mathsf{Core}_i^{\mathsf{epoch}}$ will contain at least $n-t$ parties.

Delivery

From the *Flooding* property we know that if x is given to the flooding system at a correct process, then x will eventually end up in $\mathsf{Received}_i$ of all correct P_i. For the x to be delivered we actually only need that x ended up in at least $t+1$ sets $\mathsf{Received}_i$ of correct P_i. Let epoch be an epoch at which this has already happened before the epoch starts. From *Progress* we know that all correct P_i will eventually reach epoch epoch. If x has already been delivered at this point, we are done. Otherwise we will have that $x \in U_j^{\mathsf{epoch}} = \mathsf{Received}_j \setminus \mathsf{Delivered}_j$ for at least $t+1$ honest P_i. Since $\mathsf{Core}^{\mathsf{epoch}}$ has size at least $n-t$, at least one of the P_j with $x \in U_j^{\mathsf{epoch}}$ will be in the core, which implies that $x \in V_i^{\mathsf{epoch}} = \cup_{\mathsf{P}_j \in \mathsf{Core}} U_j^{\mathsf{epoch}}$.

We see that any x input to a correct P_i will eventually end up in V_i^{epoch} at all correct P_i, at which point it is delivered. The time it takes is the time for x to reach at least $n-t$ parties and then for one more epoch to pass.

11.7 Error Detection and Group Change

We have so far analysed the protocols assuming that the set of servers is static and that some of the servers are corrupted and some of correct. However, in practice a more realistic setting is that when some server is detected to be corrupted, for instance crashed, then it will be removed from the system and a new server will be added. There might also be other good reasons to remove or add servers. Such changes are called group change. The challenge is to let all servers in the system agree on who is part of the system and to add and remove servers while the system is running. We will investigate how to do this with the above asynchronous system. Many of the techniques apply to other concrete systems. We will look at the eviction procedure and the entry procedure in isolation. Their operation does not depend on each other.

11.7.1 Error Detection

We will assume that there is a subsystem CorruptionDetection which allows to detect which servers are corrupted. It is a system for some servers $\mathsf{S}_1, \mathsf{S}_2, \ldots$. It has the following protocol ports:

- A port Accuse$_i$ where a server S_i can accuse some other server S_j of being corrupted. The input is of the form $(\text{IsCorrupt}, \mathsf{S}_j, \mathsf{epoch})$. This can be due to seeing messages from S_j that it should not send or it could be due to S_j being excessively slow (for instance, it did not send a message for a week and the system administrator does not pick up her telephone.) All other sub-system have access to accusing.
- A port Report$_i$ on which the system reports to S_i whether S_j has been detected to be corrupted. The output is of the form $(\text{IsCorrupt}, \mathsf{S}_j, \mathsf{epoch})$.

We make the following requirements:

Eventual Agreement: If an honest server outputs $(\text{IsCorrupt}, \mathsf{S}_j, \mathsf{epoch})$ then all correct servers eventually output $(\text{IsCorrupt}, \mathsf{S}_j, \mathsf{epoch})$.

Validity: If an honest server outputs $(\text{IsCorrupt}, \mathsf{S}_j, \mathsf{epoch})$ then at some earlier point in time an honest server input $(\text{IsCorrupt}, \mathsf{S}_j, \mathsf{epoch})$.

No False Positives: If an honest server inputs $(\text{IsCorrupt}, \mathsf{S}_j, \mathsf{epoch})$, then S_j is corrupt.

Note that the first property is a liveness property and the second is a safety property. The third property is something very different and new, it is namely a requirement on the parties *using* the system. It says that if you are honest, then you only accuse corrupted parties. We call this a contract property. The way to understand such properties are as follows. If you are the user of the system, then the systems promises to have all its liveness and safety properties as long as you fulfil all the contract properties. The moment you break a contract property, all bets are off.

There are two aspect to implementing the above system.

1. How do we determine when it is safe to accuse a server?
2. How do we implement the internal logic of the system, i.e., how does the accusations result in other servers outputting detections.

Another way to see this distinction is that first a server has to locally determine whether or not some other remote S_i is corrupted. After that so kind of global decision has to be made on this.

Determining Globally when a Server is Corrupted

One way to implement the internal logic of the system is a follows. If S_i gets input $(\textsc{IsCorrupt}, \mathsf{S}_j, \mathsf{epoch})$, then it signs $(\textsc{IsCorrupt}, S_j, \mathsf{epoch})$ and sends $(\textsc{IsCorrupt}, \mathsf{S}_j, \mathsf{epoch})$ along with the signature to all other servers. If a server sees $t+1$ correctly signed $(\textsc{IsCorrupt}, S_j, \mathsf{epoch})$ for the same P_j it outputs $(\textsc{IsCorrupt}, \mathsf{S}_j, \mathsf{epoch})$. Then it relays the $t+1$ signatures to make the other servers make the same output, which gives the eventual agreement. If we consider a system with at most t corrupted servers, then validity follows from the fact that one only does detection after seeing $t+1$ accusations.

The epoch is a counter which helps protect against the following attack. Assume S_1 was crashed and detected as crashed by $t+1$ signed accusations. Then it is later brought back to life, maybe by the system administrator turning the machine off and on again. When it is back up an adversary could use the same old $t+1$ signed accusations to evict it again. Therefore the counter: the intended use is that when a server is brought back to life it is given a new epoch identifier. Only signed accusations with that new epoch in them can be used to accuse the new instance of the server.

Determining Locally when a Server is Corrupted

Determining when to accuse a server in the above system is a bit more tricky. One way could be to use a number of heuristics for detecting whether a system is crashed. If for instance you did not hear from a server for a while, you might call the system administrator and ask if the system crashed. If he confirms, then it would be safe to input $(\textsc{IsCorrupt}, \mathsf{S}_j, \mathsf{epoch})$. Another approach would be to use a $\mathsf{heart\ beat}$ system where live servers send out a message every minute. Then you set a huge timeout, for instance three hours. If you did not hear from a server in three hours you call it corrupted. One objection here should be: *But you said we are in the asynchronous model! So you can't use time!* This is true. But it is common to use time in some mild way to do error detection. Still, to avoid fragile system designs we do not want to use timeouts throughout the system. It therefore makes sense to encapsulate the use of time into a sub-system as the above which takes care of $\mathsf{error\ detection}$ and then design the rest of the system without using timeouts. One advantage of this is that the overall system does not deadlock due to the use of the timeouts: the asynchronous part of the system keeps running even if some servers are slows. The timeout are only used to detect if some server is down. Therefore we can set the timeouts high: we just need to detect and replace corrupted servers faster than they crash.

There are of course some more low hanging fruits too in detecting errors. If for instance a server has a bug and starts sending wrong messages, then you can accuse the system. When $t + 1$ servers received wrong messages from the faulty server it would eventually be detected and replaced.

There is an important take-home message above, so it is worth saying it loud once more:

Using time only for error detection and designing the rest of the system in the fully asynchronous model allows to use timeouts in a way that does not slow down the system during normal operation.

11.7.2 Eviction

Eviction of corrupted servers proceeds as follows:

- On output $(\textsc{IsCorrupt}, \mathsf{S}_j, \mathsf{epoch})$ on $\textsc{CorruptionDetection}.\mathsf{CD}_i$, send
$$(\textsc{IsCorrupt}, \mathsf{S}_j, \mathsf{epoch})$$
on $\mathrm{TOB}.\mathrm{TOB}_i$.
- If $t + 1$ servers sent $(\textsc{IsCorrupt}, \mathsf{S}_j, \mathsf{epoch})$ on TOB then ignore all messages from S_j in all protocols. Simply do as if S_j stopped sending messages. At the same time, stop sending any messages to S_j over point-to-point channels.

Notice that this eviction procedure is secure in any system tolerating Byzantine errors. Namely, we only evict S_j if it is corrupted, and if S_j is corrupted, then not sending any further messages is included in the possible malicious behaviour of S_j, so the system can already handle that S_j stops sending messages. Also, not sending messages on the point-to-point channels does not affect the security of the system. As a corrupted party S_j could just have claimed to not having received these message anyway.

11.7.3 Entry

We would also like to be able to add a new server. Consider a server at Step 1. The only information it needs to run the new epoch is epoch, $\mathsf{Received}_i$ and $\mathsf{Delivered}_i$. This is called the $\mathsf{entry\ information}$. To start participating, it needs to get this information in a reliable way from somewhere.

It can get epoch by asking on the network. Some severs might be slow and report an old number or no number at all. Other servers might be malicious and report a wrong number, maybe one into the future. It turns out that it is safe to adopt a too low number but not advisable to pick a too high one. The reason why it is secure to adopt a too low number is that it just means that you will do reentry from an earlier point and then have to run for a bit longer to catch up. Picking a too large number would have the entry being postponed into the future, and for the sake of security we would like the reentry to happen as fast as possible to we can get back to a maximal number of correct servers again. So if there are at most t corrupted servers, then the following will do:

- S_i: Broadcast $(\textsc{Entry}, \mathsf{S}_i)$ on the totally-ordered broadcast.
- When S_j sees $(\textsc{Entry}, \mathsf{S}_i)$, it makes a connection to S_i and sends epoch_j.
- S_i: Collect epoch_j from $n - t$ parties. Remove the t largest numbers and adopt the largest one among the remaining ones.

Now S_i knows an epoch epoch that some honest parties have already past. So they will all eventually make it to that epoch, and when they do they will agree on $\mathsf{Delivered}$. So the following will let S_i learn $\mathsf{Delivered}$.

- S_i: Broadcast $(\textsc{Entry}, \mathsf{S}_i, \mathsf{epoch})$.
- When S_j sees $(\textsc{Entry}, \mathsf{S}_i, \mathsf{epoch})$, it makes a connection to S_i and sends $\mathsf{Delivered}_j^{\mathsf{epoch}}$: the value that S_j had for $\mathsf{Delivered}$ at the end of epoch epoch.
- S_i: Collect $\mathsf{Delivered}_j^{\mathsf{epoch}}$ from $2t + 1$ parties and adopt the value that was sent by at least $t + 1$ servers.

Now S_i knows a valid epoch epoch and the set $\mathsf{Delivered}$ at the end of that period. It sets $\mathsf{Delivered}_i = \mathsf{Delivered}$. Notice that it is perfectly possible for S_i to have $\mathsf{Received}_i = \mathsf{Delivered}_i$ at the end of an epoch if all other messages were slow. So S_i can now set $\mathsf{Received}_i = \mathsf{Delivered}_i$ and be in a state which would have been possible if S_i had been correct all the time but just received some message late. It is therefore now a fully operational correct node again.

Now S_i has a legal state for a server having made it to epoch epoch and not having received any messages after the epoch yet. So now we need a way for S_i to enter the flooding network and receive all messages that were sent in epoch epoch or later. This is done as follows: enter the flooding network to start receiving all new messages. Then ask $2t+1$ peers to forward old messages and use majority to find the right ones. Now S_i adds all old and new messages to $\mathsf{Received}_i$ and stars running from state $(\mathsf{epoch}, \mathsf{Delivered}_i, \mathsf{Received}_i)$. It is now a fully operational new server.

There are many nitty-gritty details to view change that we did not cover above, like how do we make sure we do not add a new copy of S_i before the old copy is evicted by all correct processes and so on. In this book we will not cover view change in further detail.

Exercise 11.1 (Total Order by Sequencer) Start from your solution in Exercise 6.13 and make it into a system with total order using the following idea:

1. Your system runs in two phases. In phase 1 the peers connect to the network. In phase 2 they can send signed transactions.
2. The peer that started the network is a designated sequencer.
3. The sequencer creates a special RSA key pair called the sequencer key pair.

4. When connecting to a network the new client is informed who is the sequencer.
5. It is the order in which the sequencer received the transactions that counts. This is communicated to the other peers as follows: Every 10 seconds the sequencer will take the transactions that it saw, but which have so far not been sequenced. Then it puts the IDs of those transactions into a block. A block has a block number and an ordered list of IDs, `[]string`. It numbers the blocks $0, 1, \ldots$ in the order they are sent. The sequencer signs the block and sends the block on the network.
6. A client will accept a block if and only if it has the next block number it has not seen yet and the block is signed by the sequencer.
7. All clients process the transactions they receive in the order chosen by the sequencer.
8. A transaction is ignored if it would make the sending account negative.

Your solution should describe

- How your system was designed and why.
- How the TA can run your system.
- Your test must try to send transaction at the same time at different peers to see if the system handles concurrent transactions correctly. A suggestion for one test could be: Have an account with $1,000$ coins on it. Have a program P_1 which at replica 1 repeatedly executes a transaction moving 1 coin from account A to B. It should send the transaction 1000 times. Have a program P_2 which at replica 2 repeatedly executes a transaction moving 1 coin from account A to C. It should send the transaction 1000 times. Run the two programs at the same time. Make sure that all replicas see the same number of coins end up on all accounts. check that account A is 0 when it is all done. If your system is too slow to do 2000 transactions, pick a lower number. But it is important that the test runs long enough that both programs are running at the time where account A hits 0.
- How the system was tested and how to run your test if you did an automatic test.

Your do not need to:

- Handle errors, neither Byzantine nor crash errors. In particular, if the sequencer crashes, then the system is allowed to die.
- You do not have to make your test automatic, but it is recommended.

△

11.8 Client-Centric Consistency versus Server-Centric Consistency

A last problem we want to discuss with respective to state machine replication is client-centric consistency. We have so far guaranteed that each correct server will execute the same commands in the same order. This is know as server-centric consistency. Note, however, that one server S_1 might be behind another server

S_2. Maybe S_1 executed 100 commands so far, but S_2 had a productive day and executed 1000 commands. If there is a light client C in the system that is not running as a full servers, but is just reading the state of the replicated machine from one of the server, this might give problems. Such thin clients are quite common in for instance crypto currencies, where it is extremely expensive to run a full node.

Consider a client that has been reading its state from S_2. Now it changes to S_1. Maybe because S_2 crashed or maybe because C is mobile and changed to S_1 because S_1 is now closer to C. If this happens, then C will suddenly be reading an earlier state of the replicated machine: the state after 100 commands being executed as opposed to the state after 1000 commands being executed. Traveling back in time could be very confusing for the client.

Another possibility is that some of the servers might be malicious. Assume that C reads from server S_1, but S_1 is malicious. Then S_1 could simply report a wrong state to C. This could be dangerous in a cryptocurrency system as S_1 could trick C into falsely believing it received some funds.

Exercise 11.2 (Client-Centric Consistency I) Assume that all servers are honest and server-consistent, but might drift apart in how many commands they executed. Consider a client that wants to move from S_1 to S_2. Which of the following methods work:

1. Log off S_1, wait for an hour and then start reading from S_2.
2. Before moving ask S_1 how many commands it executed. Call this number c. When logging onto S_2, ask it how many commands it executed. Wait until it executed at least c commands, then start reading from S_2.
3. Before moving broadcast a big random number N on the broadcast channel via S_1. When logging onto S_2, wait until N arrives at S_2, and then start reading from S_2.

$\triangle$

It is often desirable that a client does not need to keep state between sessions. The next exercise asks you to think about a system that achieves that even when there are corrupted servers.

Exercise 11.3 (Client-Centric Consistency II) Assume that there are n servers and that t can be maliciously corrupted and $n = 4t + 1$. The honest servers are server-consistent, but might drift apart in how many commands they executed. Which of the following methods allow clients to read with client consistency:

1. When the client C wants to read the state of the system it reads from any t servers and takes the latest state reported.
2. When C wants to read the state of the system it reads from any $2t + 1$ servers and records the larges number of commands executed by any server. Then it waits until all servers executed c commands. Then it takes the state which is reported by $t + 1$ of the servers.

3. When C wants to read the state of the system it reads from any $3t+1$ servers and records the largest number of commands executed by any server. Then it waits until all servers executed c commands. Then it takes the state which is reported by $2t+1$ of the servers.
4. When C wants to read the state of the system it reads from any $4t+1$ servers and records the largest number of commands executed by any server. Then it waits until all servers executed c commands. Then it takes the state which is reported by $3t+1$ of the servers.

△

11.8.1 More Exercises

Exercise 11.4 (State Machines) Which of the programs/services are state machines:

1. A program which on input `readDate` outputs the date of the day.
2. A program which on input x outputs $x+1$.
3. A program which on input `readTemperatur` accesses a thermometer attached to the computer it runs on, reads the current temperature t in the room it is in, and then outputs t.
4. A program which on input x runs a random number generator to get a number r and then outputs $x+r$.
5. A program which on input `readYear` outputs which year it is.
6. A program which on input $(\texttt{store}, x)$ stores x and on input `read` outputs the last x that was stored.
7. A program which on input x outputs the sum of all the number that were ever input to it.

△

Exercise 11.5 (Synchronous Totally Ordered Broadcast (I)) Consider the following protocol for the synchronous round-based model trying to implement totally-ordered broadcast. We consider $t < n/3$ Byzantine corruptions and assume the used sub-protocols are secure against that number of corruptions.

1. When a new message x arrives at a server the server broadcasts the message using unscheduled consensus broadcast UCB. Assume the UCB could output at different parties in different rounds. So, it is for instance done by signing messages and relaying signed messages.
2. In each round each server takes the messages that it received from all the other parties, orders them lexicographically and outputs them in that order.

Does it work? If it does not work, explain why. If it works, is this a better or a worse system than the one we saw in the above chapter in terms of efficiency.

△

Exercise 11.6 (Synchronous Totally Ordered Broadcast (II)) Consider the following protocol for the synchronous round-based model trying to implement totally-ordered broadcast. We consider $t < n/3$ Byzantine corruptions and assume the used sub-protocols are secure against that number of corruptions.

1. When a new message x arrives at a server the server broadcasts the message using unscheduled consensus broadcast. This rule is always active, i.e., it goes on in the background independently of the below rules.
2. In each so-called epoch, in some pre-agreed round each server takes the messages that it received from all the other parties during this epoch and broadcasts them using scheduled consensus broadcast (SCB). Say the SCB terminates at all parties in the same round to avoid that parties desynchronise. A bit more precisely: The first round of SCBs is run in round 1. The next round of SCBs is run the round after the first round of SCBs all terminated and so on. One epoch is the rounds it takes to run one round of SCB.
3. At the end of an epoch, when all the SCBs terminated, all servers takes the union of the sets output be the SCBs, orders them lexicographically and outputs them in that order. Then a new epoch starts, i.e., they go back to Step 2.

Does it work? If it does not work, explain why. If it works, is this a better or a worse system than the one we saw in the above chapter in terms of efficiency.

△

Exercise 11.7 (Asynchronous Totally Ordered Broadcast (I)) Consider the following protocol for the asynchronous model trying to implement totally-ordered broadcast. We consider $t < n/3$ Byzantine corruptions and assume the used sub-protocols are secure against that number of corruptions.

1. When a new message x arrives at a server the server broadcasts the message using unscheduled consensus broadcast.
2. In each "epoch" each server waits to receive a message from each of the other servers. Then it orders them lexicographically and outputs them in that order. Then a new epoch starts, i.e., they go back to Step 2.

Does it work? If it does not work, explain why. If it works, is this a better or a worse system than the one we saw in the above chapter in terms of efficiency.

△

Exercise 11.8 (Asynchronous Totally Ordered Broadcast (II)) Consider the following protocol for the asynchronous model trying to implement totally-ordered broadcast. We consider $t < n/3$ Byzantine corruptions and assume the used sub-protocols are secure against that number of corruptions.

1. When a new message x arrives at a server the server broadcasts the message using unscheduled consensus broadcast (UCB).

2. Each server then waits to receive a message from $n - t$ of the other servers. Then it broadcasts this set using UCB.
3. Each server then waits to receive these sets of message from $n - t$ of the other servers. Then it pick the set from the server with the lowest name (order the names lexicographically say). Then it orders them lexicographically and outputs them in that order.
4. Then go to Step 2.

Does it work? If it does not work, explain why. If it works, is this a better or a worse system than the one we saw in the above chapter in terms of efficiency.

△

11.9 Core-Set Selection

We saw above that one cannot solve ABA using a deterministic protocol. Recall how we did Byzantine agreement in Chapter 8 once we could do broadcast: each party would broadcast its vote, and then based on the n votes we would make a common decision. That seems like a nice and deterministic solution to Byzantine agreement! Why do we not use the same solution in the asynchronous model? The problem is that in the asynchronous model we cannot distinguish a slow broadcast from a broadcast that was never sent because the sender is malicious. The best we can hope for is to receive a broadcast from $n - t$ parties. If only we could agree on which $n - t$ broadcasts to use, we would still be done: based on the $n - t$ votes we would make a common decision. So we can conclude that we cannot solve that problem deterministically either.

In this section we implement a tool that *almost* does what we need. It is in some sense the task closest to ABA that we can solve deterministically. It goes as follows All parties will broadcast a value V_i. At the end each party will have received a lot of the values V_i. Furthermore, there will be a core which is $n - t$ parties P_i from which all honest parties received V_i. This only problem is that some parties could have received 'more than $n - t$ values, and then they do not know who the core is, just that it exist. Note that if we could ensure that all parties received only from the $n - t$ parties in the core, then we could solve ABA, so to solve Core-Set Selection deterministically we need to let go of something. Letting some honest parties receive some extra broadcasts seems like a small price to pay.

11.9.1 Eventual Justification

It turns out that it is convinient to have a version of CSS which allows to select only good value V_i. We will therefore work with a notion of a value V_i being justified for party P_i. This just means that it is a value P_i is allowed to send. Whether or not a value V_i is justified for P_i in the view of P_j might depend on the local view of P_j. It might be that we say P_i is only allowed to input a V_i

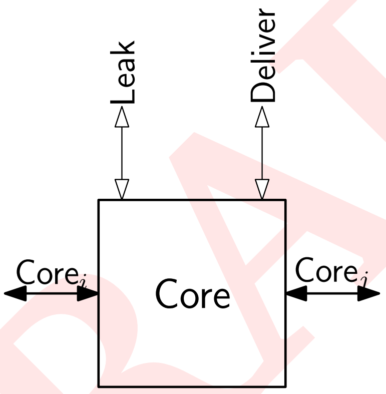
that P_i previously broadcasted using ACast. But then V_i is not justified for P_i in the view of P_j until ACast eventually outputted V_i at P_j. We will need that the notion of a value being justified will propagate: if any honest party consider a value justified, then they will all eventually consider it justified. This is true for the ACast example. To recap, we only work with notions of justification with these two properties.

Locality: Whether a value V_i is justified for a party P_i is something that any honest P_j determines from it local view state. Whether or not a value is justified can change over time.

Monotone: If P_j considers V_i is justified for a party P_i at time t, then P_j considers V_i is justified for a party P_i at times $t' > t$.

Eventual propagation: If P_j and P_k are honest and V_i is justified for P_i in the view of P_j then eventually V_i is justified for P_i in the view of P_k.

Definition 11.4 (Core-Set Selection) We define Core-Set Selection using an IF Core defined via a $\mathsf{trace\ property\ ideal\ functionality}$. We are used to this by now, so we save a bit on the formalism. It will have ports as follows:



Each Core-Set Selection will be identified by a fresh identifier cid. Each party P_i can get an input $(\textsc{Value}, \mathsf{cid}, V_i)$, where $V_i \in \{0,1\}*$ is called the value of P_i. When it is being input it must be justified for P_i in the view of P_i. Each party P_i can give an output $(\textsc{Core}, \mathsf{cid}, \mathsf{Core}_i)$, where Core_i is a set of values of the form (P_k, V_k) with each P_k occurring at most once and V_k being a bit string.

Agreement: If for some cid, two correct processes P_i and P_j output $(\textsc{Core}, \mathsf{cid}, \mathsf{Core}_i)$ and $(\textsc{Core}, \mathsf{cid}, \mathsf{Core}_j)$ and $(\mathsf{P}_k, V_k) \in \mathsf{Core}_i$ and $(\mathsf{P}_k, V_k') \in \mathsf{Core}_j$, then $V_k = V_k'$

Justified: If P_i is correct and P_i outputs $(\textsc{Core}, \mathsf{cid}, \mathsf{Core}_i)$ and $(\mathsf{P}_k, V_k) \in \mathsf{Core}_i$, then V_k is justified for P_k in the view of P_i.

Validity: If P_i and P_k are correct and P_i outputs $(\textsc{Core}, \mathsf{cid}, \mathsf{Core}_i)$ and $(\mathsf{P}_k, V_k) \in \mathsf{Core}_i$, then at some earlier point P_k had the input $(\textsc{Value}, \mathsf{baid}, V_j)$.

Large Core: When the first honest P_i outputs $(\textsc{Core}, \mathsf{cid}, \cdot)$ there exists a set Core such that $|\mathsf{Core}| = n - t$ and such that whenever an honest P_j outputs

Each P_i runs the below activation rules.

Init Let

$$\mathsf{Core}_i = \mathsf{SeenByManyHonest}_i = \mathsf{HasSeen}_i = \emptyset \ .$$

The use of the variables are as follows:

- $\mathsf{HasSeen}_i$: This is the set of parties from which P_i saw a justified value broadcast.
- $\mathsf{SeenByManyHonest}_i$: This is the set of parties P_j for which P_i knows that at least $n-t$ parties claim to have seen a justified broadcast from P_j.
- Core_i: At a sufficiently late point in the protocol Core_i will be set to be the union of the values $\mathsf{HasSeen}_j$ from $n-t$ other parties P_j. The logic of the protocol will ensure that in Core_i there will be at lot of parties that will have their broadcast seen by *all* honest parties.

Broadcast Value P_i: On input $(\text{VALUE}, \mathsf{cid}, V_i)$ where V_i is for P_i in the view of P_i, broadcast $(\text{VALUE}, \mathsf{cid}, V_i)$ using ACast.

Report Seeing Values We say that $(\text{VALUE}, \mathsf{cid}, V_j)$ has arrived at P_i from P_j when $(\text{VALUE}, \mathsf{cid}, B_j)$ was received from P_j and V_j is justified for P_j in the view of P_i. When $(\text{VALUE}, \mathsf{cid}, V_j)$ arrives from P_j, add P_j to $\mathsf{HasSeen}_i$, store $(\mathsf{P}_j, \mathsf{cid}, V_j)$, and send to all parties $(\text{SAWVALUE}, \mathsf{cid}, \mathsf{P}_j)$ via pair-wise authenticated channels.

Keep Track of Values seen by Many When having received $(\text{SAWVALUE}, \mathsf{P}_k)$ from $n-t$ distinct parties add P_k to $\mathsf{SeenByManyHonest}_i$.

When Many Values Seen by Many, Report What you Saw When it first happens that $|\mathsf{SeenByManyHonest}_i| = n-t$, send the set $(\mathsf{P}_i, \mathsf{HasSeen}_i)$ to all parties via pair-wise authenticated channels.

Compute Core We say that $(\mathsf{P}_j, \mathsf{HasSeen}_j)$ has arrived at P_i when the value was received from P_j and for each $\mathsf{P}_k \in \mathsf{HasSeen}_j$ the value (VALUE, B_k) has arrived. When $\mathsf{HasSeen}_j$ has arrived from $n-t$ parties P_j, let

$$C_i = \cup_{\mathsf{P}_j} \mathsf{HasSeen}_j \ .$$

Then output $\mathsf{Core}_i = \{\ (\mathsf{P}_j, B_j) | \mathsf{P}_j \in C_i\}$

Figure 11.4 Core-Set Selection CSS.

$(\text{CORE}, \mathsf{cid}, \mathsf{Core}_j)$ it holds that $\mathsf{Core} \subseteq \mathsf{Core}_j$. When describing the IF this is enforced by asking the adversary to input Core when it gives output to the first honest party. After that the IF refused to deliver outputs to honest P_j unless

$$\mathsf{Core} \subseteq \mathsf{Core}_j$$

.

Termination: If for some cid all correct processes got an input of the form $(\text{VALUE}, \mathsf{cid}, \cdot)$, then eventually all correct processes give an output of the form $(\text{CORE}, \mathsf{cid}, \cdot)$.

△

12

Blockchains (DRAFT)

Contents

12.1 Blockchains

In this section we first give another synchronous implementation of totally-ordered broadcast using lottery-based blockchains. It will be using time much less than than the round-based protocol and will therefore potentially be much more efficient.

12.2 Blockchains versus Cryptocurrencies

A blockchain is a currently popular way to implement totally-ordered broadcast. Cryptocurrencies can be added on top of any totally-ordered broadcast. However, the first popular cryptocurrency was Bitcoin and it was implemented on top of a blockchain-based totally-ordered broadcast, so the terms blockchain and cryptocurrency have become somewhat married in popular writing. They are, however, very different creatures. A blockchain is an implementation of totally-ordered broadcast and can be used for anything where a totally-ordered broadcast is useful. A cryptocurrency is a layer that can be put on top of any totally-ordered broadcast, not just the blockchain-based ones.

12.3 Synchrony

We assume that there is a notion of global physical time t. We assume that each party P_i has a local clock Clock_i. We assume that there is a bound $\mathsf{MaxDrift}$ on clock drift. Specifically we assume that $|t - \mathsf{Clock}_i| \leq \mathsf{MaxDrift}/2$ for all correct P_i. This in particular means that the clock of any two correct P_i and P_j will have the relation $|\mathsf{Clock}_i - \mathsf{Clock}_j| \leq \mathsf{MaxDrift}$. This could for instance be done by each client synchronoizing against a server.

12.4 Flooding System

As usual we assume that there is a flooding system. We consider a model where parties can come and go. If a party is awake when a message is sent and stays awake for long enough, then it is guaranteed to get the message. There is some fixed upper bound $\mathsf{MaxDeliveryTime}$ on the delivery time.The ideal functionality works as follows:

- For each party P_i there is a protocol port $\textsc{Flood}.\textsc{Flood}_i$ for sending and receiving messages. Each P_i also has a port $\textsc{WakeUp}_i$ and a port $\textsc{GoToSleep}_i$. There is also a special port $\textsc{Tau}$.

  There is a set Awake, which is initially empty. There is a time (seconds) parameter $\mathsf{MaxDeliveryTime}$ initially set to an arbitrary constant, say 42.

  For each P_j there is a set $\mathsf{InTransit}_j$ of messages that are in transit but were not delivered yet. It is initially empty.
- If $\mathsf{MaxDeliveryTime}' > 0$ is input on $\textsc{Tau}$ and this is the first time there is an input on $\textsc{Tau}$ and no messages have been sent yet, then update $\mathsf{MaxDeliveryTime} \leftarrow \mathsf{MaxDeliveryTime}'$. This allows the environment to set some fixed upper bound on message delivery.
- The ideal functionality keeps a set Awake. On any input on $\textsc{WakeUp}_i$ it adds i to Awake. On any input on $\textsc{GoToSleep}_i$ it removes i from Awake.
- On input x on $\textsc{Flood}_i$ where $i \in \mathsf{Awake}$, add x to $\mathsf{InTransit}_j$ for all P_j.
- On input x on $\mathsf{Deliver}_i$ where x is in $\mathsf{InTransit}_j$, output x on $\textsc{Flood}.\textsc{Flood}_i$.

For a given message x, let I_x be the time t when x was input to a correct and alive P_i the first time. Let O_x be the time t when x was delivered at the last correct P_i that was in Awake since time I_x. We assume that it never happens that $O_x - I_x > \mathsf{MaxDeliveryTime}$. For the case where x was input to a incorrect P_i the guarantee is as follows: If any correct P_i outputs x at time I_x and P_j is correct and alive when this happens and P_i and P_j remain correct and alive for $\mathsf{MaxDeliveryTime}$ seconds, then P_j will also output x no later than $\mathsf{MaxDeliveryTime}$ seconds after P_i.

Below we will also consider a weaker liveness property where it is only guaranteed that most messages are delivered in $\mathsf{MaxDeliveryTime}$ seconds most of the time. We will make this more precise below.

12.5 Lottery System

We also assume a lottery system LOTTERY. It breaks time up into so-called slots of length $\mathsf{SlotLength}$. A party P_i is in slot slot if

$$\mathsf{slot} - 1 \le \frac{\mathsf{Clock}_i}{\mathsf{SlotLength}} < \mathsf{slot} \; .$$

There is a lottery system which allows a party to get a draw $\mathsf{Draw}_{i,\mathsf{slot}}$ in each slot. Each such draw has an associated value $\mathsf{Val}(\mathsf{Draw}_{i,\mathsf{slot}})$. The winner of the slot is the party with highest $\mathsf{Val}(\mathsf{Draw}_{i,\mathsf{slot}})$. In a bit more details, the lottery proceeds as follows:

- Each party has a associated value $\mathsf{Tickets}_i$. Think of this as how many tickets P_i has bought. These numbers are known by all parties.
- In the first round each party generates a key pair $(\mathsf{vk}_i, \mathsf{sk}_i)$ for a signature scheme and vk_i is broadcast to all other parties. This signature scheme should have unique signatures such that for each vk_i and each message m there is at most one value σ such that $\mathsf{Ver}_{\mathsf{vk}_i}(\sigma, m) = \top$. This holds for instance for the RSA signature scheme, of the keys were generated honestly.
- In the second round a random number Seed is made public. It should be unpredictable by all parties in round 1. We will discuss later how such a random number could be picked.
- For each $\mathsf{slot} = 0, 1, \ldots$, party P_i can compute the draw

$$\mathsf{Draw}_{\mathsf{slot},i} = \mathsf{Sig}_{\mathsf{sk}_i}(\text{LOTTERY}, \mathsf{Seed}, \mathsf{slot}) \; .$$

- The value of a draw is defined as follows: If $\mathsf{Ver}_{\mathsf{vk}_i}(\mathsf{Draw}, (\text{LOTTERY}, \mathsf{Seed}, \mathsf{slot})) = \bot$, then $\mathsf{Val}(\mathsf{P}_i, \mathsf{slot}, \mathsf{Draw}) = -\infty$. Otherwise,

$$\mathsf{Val}(\mathsf{P}_i, \mathsf{slot}, \mathsf{Draw}) = \mathsf{Tickets}_i \cdot H(\text{LOTTERY}, \mathsf{Seed}, \mathsf{slot}, \mathsf{P}_i, \mathsf{Draw}) \; .$$

Who Wins?

In each slot the winner is the node P_i with the highest $\mathsf{Val}(\mathsf{P}_i, \mathsf{slot}, \mathsf{Draw})$.

The function H will be a cryptographic hash function with 256-bit output, for instance SHA256. For the analysis we will assume that it is a $\mathsf{random\ oracle}$, that is, it output uniformly random values (except that in the same input it always outputs the same value). We think of these bit strings as number between 0 and $2^{256} - 1$. Since the outputs of H are assumed to be uniformly random in $[0, 2^{256})$ the probability that two draws will ever have the same value is negligible. Therefore each slot has a unique winner except with negligible probability. However, to be completely sure we have a unique winner, let us that that if two parties P_i and P_j have $\mathsf{Val}(\mathsf{P}_i, \mathsf{slot}, \mathsf{Draw}_i) = \mathsf{Val}(\mathsf{P}_j, \mathsf{slot}, \mathsf{Draw}_j)$, then we simply say that $\mathsf{Val}(\mathsf{P}_i, \mathsf{slot}, \mathsf{Draw}_i) < \mathsf{Val}(\mathsf{P}_j, \mathsf{slot}, \mathsf{Draw}_j)$ when $\mathsf{vk}_i < \mathsf{vk}_j$ in the lexicographic ordering. We use the short hand

$$\mathsf{Draw}_{\mathsf{slot},i} \leftarrow \mathsf{Draw}(\mathsf{sk}_i, \mathsf{slot})$$

for P_i computing its draw in slot slot.

What do you Win?

Recall that when we looked a state-machine replication and synchronous totally-ordered broadcast the blueprint for the totally-ordered broadcast was to use a flooding system to disseminate the messages and then in each round have a leader send a block which determined the order of some of the messages that had not so far been ordered. We used a simple round-robin schedule to elect the leader. Blockchain-based totally ordered broadcast is meant for a peer-to-peer system with sporadic participation (nodes come and go). In such a system we cannot use round-robin as we don't know which nodes are present in an hour from now and might not agree on who is alive at any given point in time. What we will do instead is use the above lottery system to elect the next leader.

Example: Cryptomoney as Tickets

In the above description the number of tickets is just an abstract number. There are many different choices of how to determine this numbers. One could for instance just give each P_i one vote. Then all nodes get to be leader with about the same frequency. One could also give more tickets to nodes that have shown to be online a lot, as it does not make sense to elect a leader that is not online to send the next block.

A very popular way to define the number of tickets a node P_i has in cryptocurrencies is to say that the number of tickets is proportional to be about of money on the account of P_i. The reason for this choice is that an entity with more value stored in a system presumably would be more interested in the system being healthy: few people would burn down the bank in which their own money is stored. Therefore it makes sense to have nodes with more money stored in the system get to be leaders more often. This is known as proof-of-stake blockchain systems. For concreteness the reader can therefore in the following think of a blockchain system used to implement a cryptocurrency, where to each vk_i an amount of cryptomoney $a(\mathsf{vk}_i)$ is associated. Then think of $\mathsf{Tickets}_i = a(\mathsf{vk}_i)$.

Winner Private until Announced

Notice that only the party P_i who knows sk_i can compute

$$\mathsf{Draw}_{\mathsf{slot},i} = \mathsf{Sig}_{\mathsf{sk}_i}(\textsc{Lottery}, \mathsf{Seed}, \mathsf{slot}) \ ,$$

otherwise the signature scheme would not be secure. Therefore $\mathsf{Val}(\mathsf{P}_i, \mathsf{slot}, \mathsf{Draw}_i)$ is completely unknown to all other parties until P_i makes $\mathsf{Draw}_{\mathsf{slot},i}$ public. This means that a possible network attacker does not know who the next round leader is until the leader makes $\mathsf{Draw}_{\mathsf{slot},i}$ public. This is an important design principle to withstand so-called deniable of service (DoS) attacks, where an attacker tries to attack and crash important servers in the system. Resilience to DoS attacks is particularly important in an open peer-to-peer system where the servers do not live behind some shared firewall that can keep the attacker out. Since all the peers can send messages to each other, so can the attacker. Often sending a lot of messages to a machine is all it takes to crash it. So, if an attacker knew all future round leaders well ahead of time it could focus on crashing these before it is their time to lead. This would kill the liveness of the system.

Winner can be Recognised

Notice, however, that all parties know all the values $\mathsf{Tickets}_i$ and vk_i, so all parties can compute $\mathsf{Val}(\mathsf{P}_i, \mathsf{slot}, \mathsf{Draw})$ once they are given $(\mathsf{P}_i, \mathsf{slot}, \mathsf{Draw})$. Hence the winner can be recognised once it makes itself known.

Why Unique Signatures?

The reason why we need unique signatures is that if P_i could compute several valid signatures for $(\textsc{Lottery}, \mathsf{Seed}, \mathsf{slot})$, then a malicious server would get multiple attempts at winning the lottery, and we would like that corrupt processes lose the lottery as often as possible.

What is the Seed About?

Assume that Seed was not there. Then we would have that

$$\mathsf{Draw}_{\mathsf{slot},i} = \mathsf{Sig}_{\mathsf{sk}_i}(\textsc{Lottery}, \mathsf{slot}) \ ,$$

so the future draws would essentially depend only on slot. Then when a corrupted P_i is picking $(\mathsf{vk}_i, \mathsf{sk}_i)$ it could try to pick it to make $\mathsf{Val}(\mathsf{P}_i, \mathsf{slot}, \mathsf{Sig}_{\mathsf{sk}_i}(\textsc{Lottery}, \mathsf{slot}))$ as large as possibly for a lot of future values of slot, to get an advantage in the lottery. The use of Seed thwarts this attack as vk_i is picked before Seed is made public.

12.6 A Protocol that Almost Works

For illustration we first consider a protocol where all parties participate all the time. The protocol is described in Fig. 12.1.

Let us first look at what works and then later look at the parts that do not work.

Uses: • Partially synchronous clocks slot_i as described above.
• A synchronous flooding network FLOOD.
• A lottery system LOTTERY.
Implements: Totally ordered broadcast.
Fault types: Byzantine corruptions of at most 33% of the tickets.

The protocol runs as follows:

- P_i : Let slot_i be the slot number. Initially, let $\mathsf{Delivered}_i = \emptyset$. Let $\mathsf{Received}_i$ be the messages received by the flooding system.
- Each P_i runs the below activation rules. All values are sent via FLOOD.
  1. As the first thing in the new slot, get the draw Draw_i, let $U_i^{\mathsf{slot}} = \mathsf{Received}_i \setminus \mathsf{Delivered}_i$ and send $V_i = (\text{BLOCK}, \mathsf{slot}, \mathsf{Draw}_i, U_i^{\mathsf{slot}})$ along with $\sigma_i = \mathsf{Sig}_{\mathsf{sk}_i}(V_i)$. Let $\mathsf{cw} = i$ and store $(\mathsf{slot}, \mathsf{P}_{cw}, \mathsf{Draw}_{\mathsf{cw}}, U_{\mathsf{cw}}^{\mathsf{slot}})$ as the current winner of this slot.
  2. Until the end of slot slot collect incoming values $(V_j = (\text{BLOCK}, \mathsf{slot}, \mathsf{Draw}_j, U_j^{\mathsf{slot}}), \sigma_j)$. If $\mathsf{Val}(\mathsf{P}_j, \mathsf{slot}, \mathsf{Draw}_j) > \mathsf{Val}(\mathsf{P}_{\mathsf{cw}}, \mathsf{slot}, \mathsf{Draw}_{\mathsf{cw}})$, then store $(\mathsf{slot}, \mathsf{P}_j, \mathsf{Draw}_j, U_j^{\mathsf{slot}})$ as the current winner and let $\mathsf{cw} = j$.
  3. At the end of slot slot, let $(\mathsf{slot}, \mathsf{P}_{cw}, \mathsf{Draw}_{\mathsf{cw}}, U_{\mathsf{cw}}^{\mathsf{slot}})$ be the current winner. Output $U_{\mathsf{cw}}^{\mathsf{slot}}$ on TOB.TOB_i in some deterministic order and let $\mathsf{Delivered}_i = \mathsf{Delivered}_i \cup U_{\mathsf{cw}}^{\mathsf{slot}}$.

Figure 12.1 A blockchain consensus protocol the does not work.

Let us first look at agreement. Assume that all parties agree on $\mathsf{Received}_i$ at the beginning of slot slot. We want to argue that then they also agree on $\mathsf{Received}_i$ at the end of the slot. In slot slot, let $\mathsf{P}_{\mathsf{win}}$ be the winner of the lottery, i.e., the party with the highest $\mathsf{Val}(\mathsf{P}_j, \mathsf{slot}, \mathsf{Draw}_j)$.

We first prover the following property:

Lemma 12.1 *Let* $\mathsf{MaxDrift}$ *be the maximal clock drift in the network and let* $\mathsf{MaxDeliveryTime}$ *an upper bound on the time it takes to deliver a message. If* $\mathsf{SlotLength} > \mathsf{MaxDrift} + \mathsf{MaxDeliveryTime}$ *then the following holds: If* $\mathsf{P}_{\mathsf{win}}$ *is honest in slot* slot*, then all correct* P_i *will have* $\mathsf{cw}_i = \mathsf{win}$ *at the end of slot* slot*. In particular, if the parties agree on* $\mathsf{Delivered}_i$ *at the beginning of slot* slot*, then the correct parties agree on* $\mathsf{Delivered}_i$ *at the end of slot* slot*.*

PROOF Let t_{win} be the time when $\mathsf{P}_{\mathsf{win}}$ begins slot slot. Let t_i be the time when P_i begin slots slot. Then $\mathsf{P}_{\mathsf{win}}$ sends V_{win} no later than $t_i + \mathsf{MaxDrift}$. Therefore P_i gets V_{win} no later than $t_i + \mathsf{MaxDrift} + \mathsf{MaxDeliveryTime} < t_i + \mathsf{SlotLength}$. So, P_i get V_{win} before slot slot is over at P_i and will therefore clearly the $\mathsf{cw} = \mathsf{win}$. This means that in the last step all correct parties compute $\mathsf{Delivered}_i = \mathsf{Delivered}_i \cup U_{\mathsf{cw}}^{\mathsf{slot}}$. □

Note in particular that when $\mathsf{P}_{\mathsf{win}}$ is correct, then $\mathsf{Delivered}$ grows with all the values that reached $\mathsf{P}_{\mathsf{win}}$ before the beginning of slot slot. So the system is live in the sense that it does not take a message x longer to be delivered than it takes to reach all parties and then a correct process to win.

12.6.1 The Problem

The problem with the protocol so far happens when $\mathsf{P}_{\mathsf{win}}$ is corrupted. A corrupted $\mathsf{P}_{\mathsf{win}}$ can play the following nasty trick. In stead of sending V_{win} at the beginning of slot slot it waits until for instance $t_{\mathsf{win}} + \mathsf{MaxDeliveryTime}/2$, i.e., it sends it about halfway into the slot. As a result some parties might receive V_{win} and set $\mathsf{cw}_i = \mathsf{win}$ and some might not have time to receive V_{win} and will therefore set $\mathsf{cw}_i = \mathsf{rup}$, where P_i is the $\mathsf{runner\ up}$, i.e., the parties with the seconds highest value if its draw in slot slot. As a consequence agreement is lost.

12.6.2 Creating more Problems with Optimisations

Before we fix the identified problem, we will create even more problems of the same type. We will introduce two important optimizations. Both of them create more problems of the type identified above. However, the mechanism we introduce to solve the problem identified above will also solve the problems created by the optimizations, so we can so to say introduce them for free.

Note that if the problematic situation discussed above occurs, where the slot winner is corrupted and sends the block late, then some correct parties might have different last blocks that they saw.

Agressive Round Length

Recall that it is important for the protocol that

$$\mathsf{SlotLength} > \mathsf{MaxDrift} + \mathsf{MaxDeliveryTime}\ .$$

It turns out that the tree protocol we introduce below can tolerate that

$$\mathsf{MaxDrift} + \mathsf{MaxDeliveryTime} > \mathsf{SlotLength}$$

if only it does not happen too often. This is important for efficiency, as there will be some variation in the drift. If one has to set $\mathsf{MaxDeliveryTime}$ such that it holds except with probability 2^{-80} that the delivery time is never less than $\mathsf{MaxDeliveryTime}$ throughout the lifetime of the system, it has to be set very high. If on the other hand we only have to set it such that with probability 95% it holds that the delivery time is less than $\mathsf{MaxDeliveryTime}$ at every given time slot, then we can set it much lower, which will allow to have a faster system. So for now imagine that we set $\mathsf{MaxDeliveryTime}$ such that with probability 95% it holds that the delivery time is less than $\mathsf{MaxDeliveryTime}$.

Only Send Draws with a Chance to Win

Another important optimization allows to save a lot of bandwith. In the above protocol each party sends a block to each other party in each slot, which creates a lot of traffic. We will introduce a hardness threshold $\mathsf{Hardness}$. Only parties with a ticket with value higher than $\mathsf{Hardness}$ will send its block. If we set $\mathsf{Hardness}$ high enough this will ensure that only a few blocks are sent to all parties in each slot, giving a dramatic optimization in bandwidth. However, as a consequence it can also happen that in some round no block is sent, because all tickets have a value less than $\mathsf{Hardness}$.

12.7 Growing a Tree

We now have two types of problems to solve. Sometimes a winning block might get delivered only to some correct parties—either because the block is sent late with malice or because the delivery time is too high in the slot—and sometimes there is no slot winner. It turns out that the solution to both of these problems is to grow a tree instead of a chain. Each party is then supposed to take the longest path in the tree as its chain. As we will see, this will tend to converge on agreement.

Definition 12.2 (block tree) A node is of the form $(\textsc{Block}, \mathsf{P}_j, \mathsf{slot}, \mathsf{Draw}, U, h, \sigma)$, where $\textsc{Block}$ just specifies the type of the tuple, that it is a block, $\mathsf{slot} \in \mathbb{N}$ is a $\mathsf{block\ number}$, Draw is the draw that was used to win the lottery, U is the block data, and h is a block hash (of some previous block). We define the value of a block N to be $\mathsf{Val}(N.\mathsf{P}, N.\mathsf{slot}, N.\mathsf{Draw})$. There is a special block $(\textsc{Block}, \perp, 0, \perp, U_0, \perp)$ with value ∞ called the $\mathsf{genesis\ block}$, where U_0 is the $\mathsf{genesis\ data}$. The genesis block contains $\mathsf{Hardness}$ and more. We sometimes also just call G $\mathsf{genesis}$. A set S of blocks which contains G and where all blocks are valid defines a tree as follows. The nodes of the tree is a sub-set of the nodes in Tree. The root of the block tree is the genesis block G. Edges are directed and points towards the root. There is an edge to $N_1 \in \mathsf{Tree}$ from $N_2 \in \mathsf{Tree}$ if and only if $N_2.h = H(N_1)$ and $N_2.\mathsf{slot} > N_1.\mathsf{slot}$. For a given node N we let $\mathrm{PathTo}(N)$ be the list of nodes from G to N, including G and N and indexed from 0.

△

An important component in blockchains is the notion of some paths in the tree being better than another. In general we just like the longest path the best. This is the one we want to build on. However, when two paths have the same length we might want to use a tie breaker, and when we discuss finalization later, we will add more bells and whistels.

Definition 12.3 (path weight) A blockchain is a block tree where each block has at most on parent. A path weight is a function PathWeight mapping blockchains into a totally ordered set. It should have the property that if P' is a proper prefix of P, then $\mathrm{PathWeight}(P') < \mathrm{PathWeight}(P)$. It should also have the property that if $P \neq P'$ then $\mathrm{PathWeight}(P) \neq \mathrm{PathWeight}(P')$. Finally it should have the property that it cannot happen that $\mathrm{PathWeight}(P') < \mathrm{PathWeight}(P)$ and $\mathrm{Len}(\mathrm{PathWeight}(P')) > \mathrm{Len}(\mathrm{PathWeight}(P))$. The default path weight is just the order which sorts first on length of P and then $\mathsf{Val}(\mathrm{Leaf}(P))$. So $\mathrm{PathWeight}(N_1) < \mathrm{PathWeight}(N_2)$ if and only if $\mathrm{Len}(N_1) < \mathrm{Len}(N_2)$ or $\mathrm{Len}(N_1) = \mathrm{Len}(N_2)$ and $\mathsf{Val}(N_1) < \mathsf{Val}(N_2)$.

△

Definition 12.4 (best path, best leaf) Let $\mathsf{Tree} = \mathrm{BlockTree}(S)$ be a block tree. The best path in Tree is the path from G to a leaf that maximizes PathWeight. We write $P = \mathrm{BestPath}(\mathsf{Tree})$. The best leaf of Tree is

$$\mathrm{BestLeaf}(\mathsf{Tree}) = \mathrm{Leaf}(\mathrm{BestPath}(\mathsf{Tree})) .$$

△

The protocol uses two auxiliary function GetMetaData and ValidMetaData. The first one will ask the local system if there is extra data to include in the block data. The latter will check whether some meta data is valid. For now just assume that $\mathrm{GetMetaData}() = \emptyset$ and $\mathrm{ValidMetaData}(\cdot) = \top$. The protocol proceeds as follows:

- P_i : Throughout the protocol let slot_i denote the current slot number. Initially $\mathsf{slot}_i = 0$. Initially, let $\mathsf{Delivered}_i = \emptyset$. Let $\mathsf{Received}_i$ be the messages received by the flooding system. Let G be the genesis block and initially let $S_i = \{G\}$. Throughout the protocol let $\mathsf{Tree}_i = \mathsf{Tree}(S_i)$.
- Each P_i runs the below activation rules. All values are sent via Flood.

1. Immediately when slot slot_i is entered, let $P = \mathrm{BestPath}(\mathsf{Tree}_i)$, get the draw Draw_i. If $\mathsf{Val}(\mathsf{P}_j, \mathsf{slot}, \mathsf{Draw}) \geq \mathsf{Hardness}$, then proceed as follows. Let $M_i = \mathrm{GetMetaData}$, let $U_i = \mathsf{Received}_i \setminus \mathsf{Delivered}_i(P)$, let $B_i = \mathrm{Leaf}(P)$, let $h_i = H(B_i)$, let $\sigma_i = \mathsf{Sig}_{\mathsf{sk}_i}(V_i)$, and send $V_i = (\textsc{Block}, \mathsf{P}_i, \mathsf{slot}_i, \mathsf{Draw}_i, (U_i, M_i), h_i, \sigma_i)$.
2. On input $V_j = (\textsc{Block}, \mathsf{P}_j, \mathsf{slot}, \mathsf{Draw}, (U, M), B, \sigma)$ with $\mathsf{Ver}_{\mathsf{vk}_j}(\sigma_j, V_j) = \top$ and $\mathsf{Val}(\mathsf{P}_j, \mathsf{slot}, \mathsf{Draw}) \geq \mathsf{Hardness}$, store it and wait until it happens that $\mathsf{slot}_i > \mathsf{slot}$ and $\mathrm{ValidMetaData}(M) = \top$, then add it to S_i.

12.7.1 Your are all Winners!

In the below we change the meaning of the word winner, so we dedicate this short section to avoidance of confusion. Above where all parties sent their draw we called the party with the highest value the winner, which meant that there was always exactly one winner in each slot. Now we will instead use winner to denote any party which has a value which is higher than Hardness. This means that there can be several winner and sometimes there might be no winner.

12.7.2 How does the Tree Grow

We now want to explore how the block tree behaves over time. An important concept will be the honest tree.

Definition 12.5 (honest tree) Given two valid trees Tree_i and Tree_j the union $\mathsf{Tree}_i \cup \mathsf{Tree}_j$ is simply the tree with root G that contains all the paths of both Tree_i and Tree_j. This is again a valid tree. Let H be the set of all correct P_i at time t and let Tree_i^t be the tree of P_i at this time. Let

$$\mathsf{HonestTree}_{\mathsf{slot}}^t = \bigcup_{i \in H} \mathsf{Tree}_i^t .$$

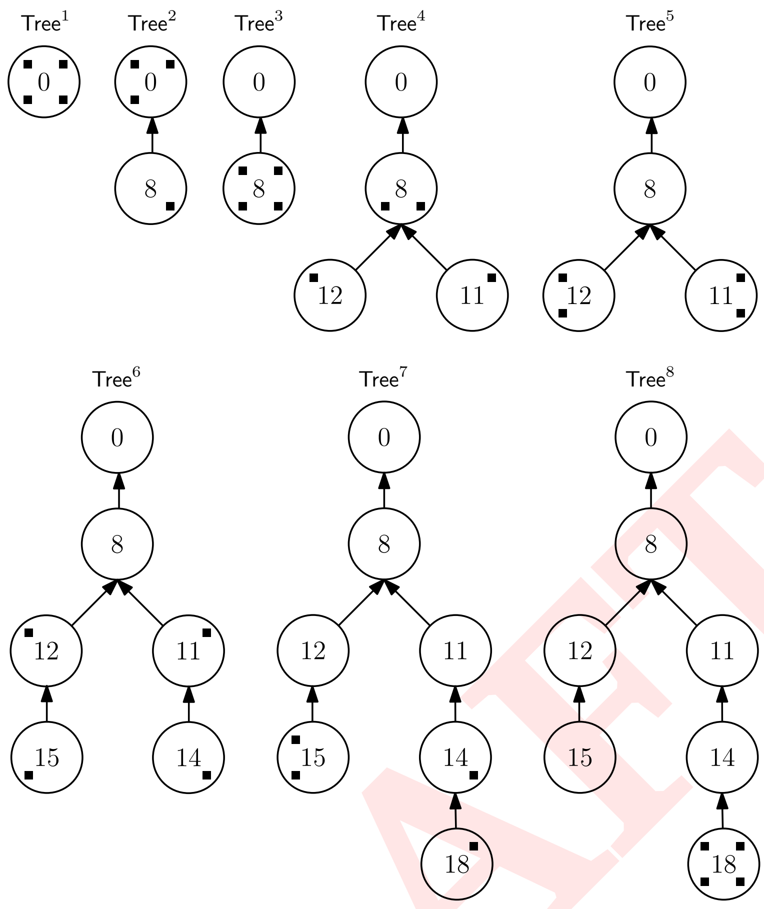


Figure 12.2 We depict four parties as small black squares. A circle around a number means that some party won that slot number and created a block. We show the whole honest tree. If we put a square in a circle, it means that that party considered that block to be the currently best leaf. At the beginning (Tree^1) all four parties know only the genesis block. Then in slot 8 some party P_i wins the slot and sends a new block. At the time P_i adds the block to S_i the honest tree grows by that block. But so far only P_i saw it. This creates Tree^2. After some time all the other processes see the new block and consider it the currently best leaf. Then some P_j wins slot 11 and right after P_k wins slot 12 before seeing block 11. This creates the situation in Tree^4. The two new blocks reach different parties at different times, creating the fifth tree. Now we get even more unlucky and one of the parties in block 11 wins slot 14 and about the same time a party in block 12 wins slot 15. They don't have time to see each other's blocks and therefore extend different paths, as shown in Tree^6. Then one of the other parties see block 15 first and the other sees block 14 first. It goes to 14 and right after wins slot 18, as shown in tree 7. Now there is a sufficiently long time where there is no new slot winner, so block 18 reach all parties before the honest tree grows. Therefore all four parties consider block 18 the best one and go to the block. Next time there is a block winner it will extend at 18 and the short so-called fork ending in block 15 will be forgotten.

△

Notice that by definition all correct parties at any time sees a tree which is a subset of the honest tree. Namely, for all correct P_i, clearly $\mathsf{HonestTree}^t = \mathsf{HonestTree}^t \cup \mathsf{Tree}_i^t$. Notice furthermore that if Δ_t is the network delivery time at time t and P_i is a party which is alive at time t until time $t + \Delta_t$, then

$$\mathsf{HonestTree}^t \subseteq \mathsf{Tree}_i^{t+\Delta_t} ,$$

as all nodes seen by any honest party at time t will reach P_i by time $t + \Delta$. So in some vague sense, if we squeeze our eyes enough that we cannot see what happens in intervals of time Δ, all honest parties basically see $\mathsf{HonestTree}$. In particular, if for long enough the honest tree does not grow, then all honest parties will in fact see the honest tree and will therefore agree on the best leaf. So if there is long enough between slot winners, all honest parties tend to build on the best path which will therefore get better and better than all small unlucky forks.

To start familiarising ourselves with how the tree grows, consider the example in Fig. 12.2. This example shows how the tree protocol tends to get rid of unlucky disagreements: at some point one of the branches of a fork will get longer than the other and reach all parties in time that they all switch to it. Notice that for this to be true, it is important that there is often enough breaks of length Δ where no new block is made. If we produce block winners much faster than the block can reach all parties, two branches as in tree 6 in Fig. 12.2 could both keep growing.

12.7.3 Rollback and Finalisation

A big problem with tree protocols is that it is not easy to know when one can safely deliver a transaction. Notice that if a party P_i delivered a transaction that was in block 12 on TOB_i, then it cannot take it back when it changes to the path $(0, 8, 11, 14, 18)$, and in that path maybe T had to be delivered at a different point in the order. Therefore agreement would be violated. When a party changes from the chain $(0, 8, 12, 15)$ to $(0, 8, 11, 14, 18)$ it is called a rollback. All the transactions performed in blocks 12 and 15 are rolled back and might now have to be performed in a different order. If we want to avoid rollbacks, parties cannot deliver on TOB_i any transaction that is in a branch that might later be rolled back. A block that cannot ever be rolled back is call a final block.

There are two different approaches to deciding when a block is final. One of them is to just to wait "long enough" and only consider a block safe when it is long enough up in the tree. Notice for instance that when the parties were divided betweens blocks 15 and 14, if they looked two blocks up in the tree, they would all see block 8, which will never be rolled back. Another approach is to run a separate process which detects which blocks are final, called a finalization layer. We will explore both approaches below.

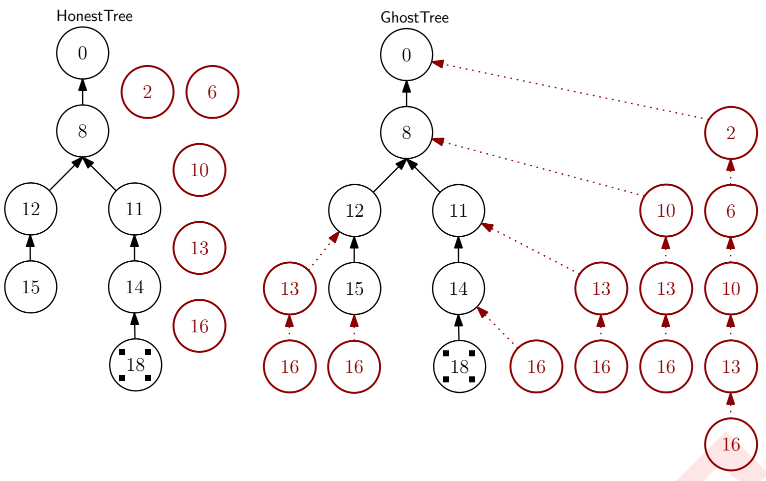


Figure 12.3 To the left: an honest tree along with five slots won by corrupted processes which did not send their block yet. To the right: some of the ghost tree. The ghost tree is just the largest tree that the corrupted parties might build with their withheld blocks. They can also choose to build some subset of the ghost tree. We only show some of the ghost tree. We only show the longest possible forks that the corrupted parties could add. They can add any sub-sequence of these. The corrupted parties could for example also choose to add $(2, 6, 13, 16)$ at 0.

12.7.4 Ghost Growth

To be able to understand how far we need to look up in the tree to find a block that will never be rolled back, we need to much better understand how the tree grows. The first step is to first look into another attack possibility by corrupted parties. Consider Tree^8 in Fig. 12.2. Recall that long break from slot 0 to slot 8, where no slot was won. It turns out that indeed slot 2 and slot 6 were won by corrupted parties. They just did not send their block. The corrupted parties also won slots 10, 13, and 16, and did not send those blocks either.

Notice that winning a slot just means that $\mathsf{Val}(\mathsf{P}_i, \mathsf{slot}, \mathsf{Draw}) \geq \mathsf{Hardness}$. So if a corrupt P_i wins slot slot it can for all U and all previous blocks B with a small slot number produce a block $(\textsc{Block}, \mathsf{P}_j, \mathsf{slot}, \mathsf{Draw}, U, H(B), \sigma)$ which will be valid. The ghost tree is the tree obtained by starting with the honest tree and for each withheld winning draw, add the corresponding blocks at all possible places. In particular, the corrupted parties can add a new block at any node in the ghost tree with a smaller slot number. Iteratively extending the ghost tree by adding all corrupted blocks at all possible places gives what is known as the ghost tree. We denote the $\mathsf{GhostTree}$ as time t by $\mathsf{GhostTree}^t$. In Fig. 12.3 some of the ghost tree is shown to the right.

If the corrupted parties produce the block corresponding to a ghost branch in the tree and send them on the network, we say that they reveal the ghost branch. Notice that if the corrupted parties reveal the ghost branch $(2, 6, 10, 13, 16)$ in the ghost tree in Fig. 12.3, then the honest parties would be forced to a long rollback, as they will all adopt $(0, 2, 6, 10, 13, 16)$ as the best path. Notice that this happened even though there were six honest winners and only five corrupt winners. However, the honest tree did not grow by seven nodes as there was some branching which wasted two honest winners. We can therefore see that if we want to avoid long rollbacks, it will be a balance between how many honest parties are in the network, how many corrupted parties are in the network, and how much wasteful branching is there is.

One solution which often comes to mind for limiting the problem is to just say that we do not allow too long rollbacks. See Fig. 12.4 to see why this is a bad idea.

12.8 Understanding Tree Growth

In understanding tree growth, three properties have crystallised out of the current scientific literature called tree growth, chain quality, and limited rollback.

12.8.1 Tree Growth

We first discuss tree growth. It basically says that the tree gets higher all the time as there are more and more slot winners. It is parametrised by a fraction TreeGrowth which says that about a fraction TreeGrowth of the slot winners make the tree get higher.

Definition 12.6 (tree growth) Let $\mathsf{TreeGrowth} \in [0, 1]$ be a real number. We say that a protocol has a tree growth of TreeGrowth if after n slot winners the height of HonestTree is at least $\mathsf{TreeGrowth} \cdot n - \kappa$ except with negligible probability $2^{-\kappa}$.

△

We show that the above protocol has a good tree growth under the assumptions that there are a lot of honest tickets, that slot winners are not produced too fast and that messages are delivered reasonably timely.

Definition 12.7 (timely slot) We define what it means for a slot slot to be timely. Let t be the time the slot starts. We say slot slot is timely if all messages sent before time $t - \mathsf{MaxDeliveryTime}$ are delivered before time t.

△

We call slot slot a honest slot if there is at least one honest winner in slot slot. An honest slot might have corrupted winners too and several honest winners. We call slot slot a lucky slot if it is a timely slot and a honest slot.

Lemma 12.8 (lucky slot) *Consider now a timely slot* slot *and let* $\mathsf{HonestTree}^t$ *be the honest tree when the slots begins. Assume that there is an honest winner in slot*

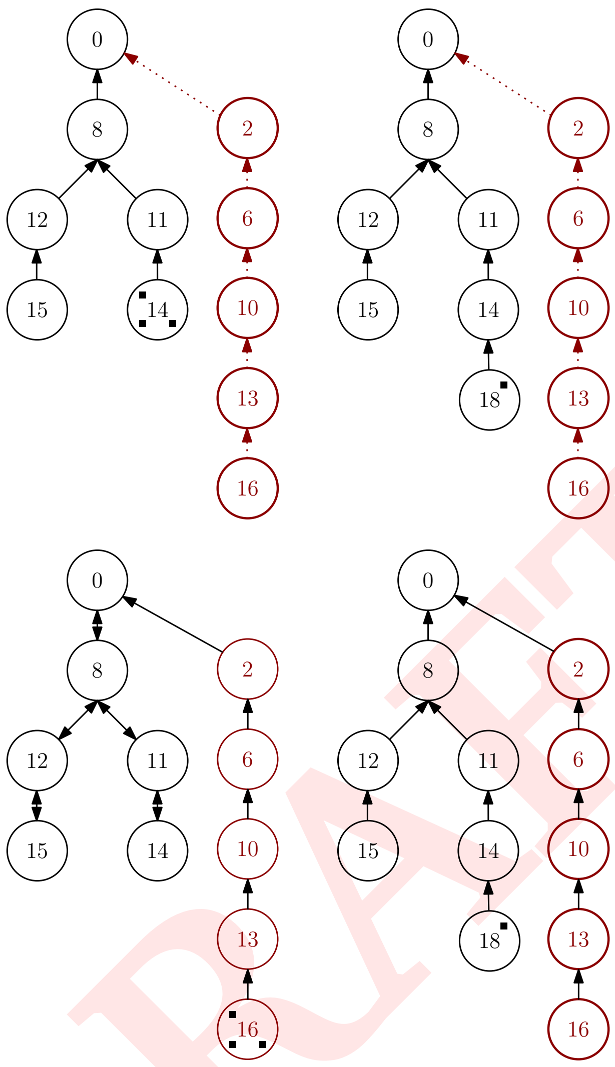


Figure 12.4 An example why it is dangerous to use time in distributed system. Imagine we say you are never allowed to do a roll back longer than three. Consider a situation as in the top row. Three parties are in block 14 still, but the slot winner already made it to block 18. Now the corrupted parties quickly reveal a ghost branch. The parties in block 14 are willing to do the roll back. The parties in 18 are not willing as the rollback is too long, so they stay on the path containing 18 forever. Now agreement will never be reached, and the network is broken.

slot. *Let* $\mathsf{HonestTree}^{t-\mathrm{MaxDeliveryTime}}$ *be the honest tree* MaxDeliveryTime *seconds before. Then*

$$\mathrm{Len}(\mathrm{BestPath}(\mathsf{HonestTree}^{t})) \geq \mathrm{Len}(\mathrm{BestPath}(\mathsf{HonestTree}^{t-\mathrm{MaxDeliveryTime}})) + 1 \ .$$

The *lucky slot* lemma say that if a slot is both timely and is being won by an honest party, then the honest tree will grow. This is easy to see. Consider the tree $\mathsf{HonestTree}^{t-\mathrm{MaxDeliveryTime}}$. Let P_i be the slot winner. By time t all messages reached P_i so $\mathsf{HonestTree}^{t-\mathrm{MaxDeliveryTime}} \subseteq \mathsf{Tree}_i^t$. Hence

$$\mathrm{Len}(\mathrm{BestPath}(\mathsf{Tree}_i^t)) \geq \mathrm{Len}(\mathrm{BestPath}(\mathsf{HonestTree}^{t-\mathrm{MaxDeliveryTime}})) \ .$$

When P_i then extends $\mathrm{BestPath}(\mathsf{Tree}_i^t)$ it gets a new tree with

$$\mathrm{Len}(\mathrm{BestPath}(\mathsf{Tree}_i')) = \mathrm{Len}(\mathrm{BestPath}(\mathsf{Tree}_i^t)) + 1 \ .$$

Since Tree_i' is part of $\mathsf{HonestTree}^t$ it follows that

$$\mathrm{Len}(\mathrm{BestPath}(\mathsf{HonestTree}^t)) \geq \mathrm{Len}(\mathrm{BestPath}(\mathsf{HonestTree}^{t-\mathrm{MaxDeliveryTime}})) + 1 \ .$$

Notice that the lemma does not say that

$$\mathrm{BestPath}(\mathsf{HonestTree}^{t-\mathrm{MaxDeliveryTime}})$$

will grow, as

$$\mathrm{BestPath}(\mathsf{HonestTree}^t))$$

might not be an extension of

$$\mathrm{BestPath}(\mathsf{HonestTree}^{t-\mathrm{MaxDeliveryTime}}) \ .$$

For instance, the adversary might reveal a longer ghost chain to move people away from $\mathrm{BestPath}(\mathsf{HonestTree}^t))$. But that ghost chain will have at least the same length as $\mathrm{BestPath}(\mathsf{HonestTree}^{t-\mathrm{MaxDeliveryTime}})$ to make parties swap to it, so the lemma still holds if P_i extends the revealed ghost chain.

By the lucky slot lemma we see that whenever there is an additional honest winner the honest tree grows in height by 1 unless one of these happened:

1. The winner is in a slot which was already honest—in which case both winners might grow the tree at the same leaf.
2. The slot was not timely (in which case the winner might not build on the best path in the honest tree).

Definition 12.9 (wasted honest winners) We call an honest slot winner wasted if it wins a slot which is not timely or it wins a timely slot which had another honest winner with a value of its ticket that is higher. We let $\mathsf{WastedHonestWinners}^t$ be the number of wasted honest slot winners at time t.

$\triangle$

This immediately gives us this result, which can be used to establish tree growth.

Theorem 12.10 (tree growth) *Let* $\mathsf{HonestWinners}^t$ *be the number of honest winners at time* t*. Then*

$$\mathrm{Len}(\mathrm{BestPath}(\mathsf{HonestTree}^t)) \geq \mathsf{HonestWinners}^t - \mathsf{WastedHonestWinners}^t \ .$$

12.8.2 The Standard Tree Scenario

We now consider an example scenario which we revisit below, so let us call it our standard tree scenario. We assume that:

- At least 66% of the tickers are honest.
- MaxDeliveryTime is such that that 95% of all slots are timely.
- The hardness such that the probability that there is a slot winner in a given slot is 10%.

We assume that the event of winning a slot is independent of whether the slot is timely. Then for a large number of slot winners there will tend to be 66% of them that are honest. Of these 5% might hit slots that are not timely, bringing the number down to about 63%. So the probability that a timely slot will have a honest winner is about 6.3% (as slots have probability 10% of having a winner at all). So the probability of hitting a slot which is already honest is upper bounded by 6.3%, so this will happen for an average of 4 out of every 63 parties. So there will be about 59% lucky slots among the slots with a winner, which means that the tree growth is 59%. For later use, notice that there will tend to be at most 34% corrupted slots.

12.8.3 Chain Quality

Chain quality asks that a lot of the nodes on the best path were produced by honest parties. In the extreme case where all nodes are produced by corrupted parties, they might censor some transactions by not adding them to their blocks. If only few nodes are produced by honest parties, the throughput of censored transactions can be lowered by the corrupted parties which can be bad enough.

Definition 12.11 (chain quality) Let ChainQuality $\in [0,1]$ be a real number. We say that a protocol has a chain quality of ChainQuality if after n slot winners the best path in HonestTree have very close to ChainQuality$\cdot L$ nodes that were produced by parties which were honest when they produced the node, except with negligible probability.

△

We can now do a very simple analysis of chain quality. Consider our standard tree scenario above. When there were n winners the honest tree will have height about $.59n$. Of these n winners about $.34n$ are corrupted, so the number of corrupted nodes on any path in the tree is at most $.34n$. Hence the longest path will have at least $.59n - .34n$ honest nodes. Since $.59 - .34 = .25$ and $.25/.59 > .42$ it follows that our protocol in the standard tree scenario has chain quality at least about 42%.

Exercise 12.1 (very good chain quality) Consider the standard tree scenario but allow yourself to assume that more tickets are honest. How big a percentage do you need to assume is honest to get a chain quality of at least 51%.

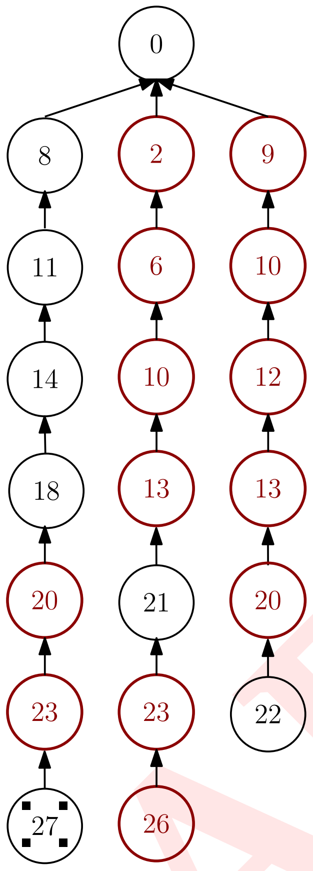


Figure 12.5 An example of paths with super slots and filler slots. The super slots are the black ones. The filler slots are the red ones. Notice that the super slots sit at different heights. This will always be the case.

12.8.4 Ghosts Eventually Die

Before proving the limited rollback property we prove that ghost chains that are kept hidden for too long will eventually become much shorter than the honest tree and therefore cannot be used for a rollback. To see this consider a ghost branch connected at slot s in the honest tree. Consider it right after it was created: it has length 1. Assume that it is still a ghost tree after 100 more slot winners were found. By then the ghost branch will tend to have grown by 34 nodes. The honest tree will tend to have grown by 59 nodes, so the ghost branch is now only $34/59 < 58\%$ as long as the honest tree from where the ghost branched off. Therefore any ghost branch will eventually fall behind. We will in a later exercise explore how fast ghost chains die off.

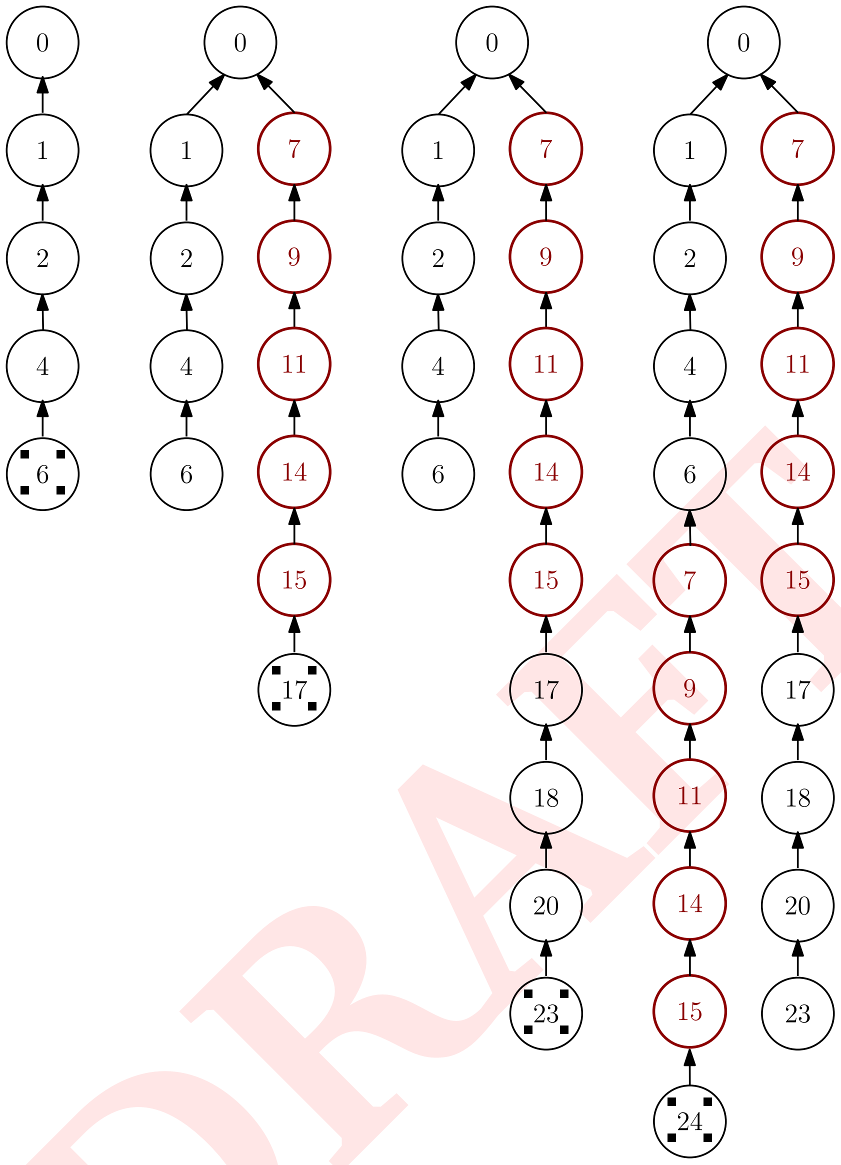


Figure 12.6 Another example of paths with super slots and filler slots. Here a rollback of length 9 is caused using only 5 distinct fillers.

12.8.5 Limited Rollback

The limited rollback property asks that there is a limit RollbackLimit such that the probability that there will ever be a rollback of length longer than RollbackLimit is some negligibly small probability.

To analyse the limited rollback property we introduce the notion of a super slot. We define a super slot to be one which was won by a honest party and where

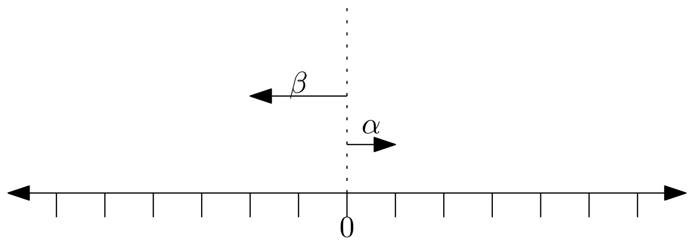


Figure 12.7 Random walk. You start from 0 and takes steps to the left with probability β and steps of some other length to the right with probability $\alpha = 1 - \beta$. Where do you tend to end up?

the slot was timely and where no other party won the slot, neither honest nor corrupted.

Consider our standard tree scenario. Out of every 100 winners, there will tend to be about $66.66\ldots$ honest ones. Less than 10% of these will hit a slot with another winner, so there will tend to be 60 that have their own slot. Of these slots about 3 will not be timely, so there will by about 57 super slots.

We call a block added in a super slot and super block.

We now prove an important lemma:

Lemma 12.12 (super slots) *All super blocks sit at different heights in the honest tree.*

To see this consider the time when super block number $i+1$ was added. At this time the honest winner saw super block number i in its tree already, as super blocks are won in different slots and in timely slots. So super block number $i+1$ was added at a higher point than super block number i.

Call all the other blocks which are not super blocks the filler blocks. Here is another important lemma:

Lemma 12.13 (filler slots) *Assume that a rollback is possible from branch B_1 to branch B_2, both rooted at node N. If since N was created there were created s super blocks, then at least $s/2$ distinct filler blocks were created.*

To get the intuition for the above theorem inspect Fig. 12.6 which shows a maximally unlucky sequence for the honest parties. In general we know that the super slots are at distinct heights. So if there are two branches involved in the rollback, then there are at least as many filler slots as super slots. Each filler block can sit on both branches, but can not sit on the same branch twice. The same logic applies even when more branches are involved, but we will not dive into the details.

To argue that we can limit roll back, our standard tree scenario is not good enough. We need to assume that 90% of all tickets are honest. So, assume that at most 10% of the winners are corrupted. Then it is easy to see that in our standard tree scenario out of every 100 slot winners about 76 will be super slots. Therefore there will tend to be only 24 filler slots. Since $2 \cdot 24 < 76$, there will tend to not be long rollbacks—there are simply not enough filler blocks to cause them. To get an even better bound on how long rollbacks we should expect consider the following game illustrated in Fig. 12.7. You start a position 0. In each round you flip a coin that comes out heads with probability 76%. If it comes out heads you take one step to the right (+1). Otherwise you take two steps to the left (−2). In one step you expected position is $.76 - 2 \cdot .24 = 0.28$. After n rounds your expected position is $0.28n$.

The question then is: how large must n be that there isn't a big chance that you position is negative?

Let ℓ be the number of steps to the left and let r be the number of steps to the right. The position is negative if and only if $r \leq 2\ell$. Since $n = \ell + r$, this is the same as $r \leq \frac{2}{3}n$. So we can consider a different game. Throw a coin as before. If it is heads go right (+1) otherwise stay put (+0). After n rounds what is the probability that your position is less than $\frac{2}{3}n$. It is well know how to analyse this using a so-called Chernoff bound. Let $X_i = 1$ if coin number i came out heads. Let Let $X_i = 0$ otherwise. Let

$$X = \sum_{i=1}^{n} X_i \ .$$

This is your position after n steps. We have that the expected value of X_i is 0.76, i.e.,

$$\mathsf{E}[X_i] = 0.76 \ ,$$

and

$$\mathsf{E}[X] = 0.76n \ .$$

We are interested in $X \geq 0.67n$. Since the expected value of X is $0.76n$ and we only need it to be $0.67n$, there seems to be hope. Since X It is the sum of n independent random variables taking on values in $\{0, 1\}$ and they are all 1 with probability 76%, the so-called Chernoff lower bound tell us that it holds for all δ with $0 < \delta < 1$ that

$$\Pr\left[X \leq (1-\delta)\mathsf{E}[X]\right] \leq e^{-\mathsf{E}[X]\delta^2/2} \ .$$

In our case $\mathsf{E}[X] = .76n$ and we are looking at the probability that $\mathsf{E}[X] \leq \frac{2}{3}n$. So we have to solve $(1-\delta)\frac{76}{100} = \frac{2}{3}$, which gives us that $\delta \geq 0.12$. We have that $e^{-.76 \cdot (.12)^2/2} < 0.995$. So plugging it all into the Chernoff bound we get that the probability of a rollback of length n is no more than 0.995^n. We can then solve for n in $0.995^n = 2^{-80}$. Taking $\log_2$ we get that $n \log_2(0.995) = -80$ or $n = -80/\log_2(0.995) \approx 11062$. So if we want security except with probability 2^{-80} we should expect rollbacks as long as 11062.

The above bound sound very pessimistic, but it can also be improved significantly. Notice in particular that we gave the adversary a lot of power for free. Inspect Fig. 12.6. To make the attack it first needs the honest parties to get ahead and then catch up with its own blocks. Otherwise all of the blocks cannot fit into both chains. In particular, to fit in both chains the blocks it uses to catch up must all have higher slot number than the super blocks the honest parties used to get ahead. Therefore in practice most filler blocks fit in only one chain. So we do not need twice as many super blocks and filler blocks. Another thing we gave for free is the row in the right most tree with nodes 7 and 15. Here there is no super node, so in the right most tree there are 8 super nodes but 10 slots that need to be filled with filler nodes, so the adversary need 5 filler nodes to compete with the 8 super nodes. In the analysis we assumed that he needed only 4. This makes no big difference when we assume a long lucky sequence, but as the expected length of lucky streaks of the adversary will be very small, these extra fillers will weigh him down significantly in practice. Finally, in the analysis we took the honest slots that did not happen to be super slots and just gave them to the adversary as if they were filler nodes. This is also too pessimistic. If for instance two honest parties both win a timely slot, then both of the nodes will be added to the tree, so the adversary cannot use one of them as a hidden ghost node. Finally, it will also happen for the adversary that it wins a slot that was already won by the adversary. This will give him no more power than winning the slot once, so the adversary too is loosing some budget to double-wins. So, all in all, there is a lot of room to improve the bound. However, proving the better bounds are significantly more complicated and we will not dive into them. Another reason to not dive too deep into them is that there is a limit to how much they will improve the bound. If we consider a scenario where at most 25% of the tickets are corrupted, then the adversary can always let one chain get ahead by 40 steps and then hope to get lucky 40 times in a row. This will happen with probability $\frac{1}{4}^{40} = 2^{-80}$, so if we want reasonable security we should never expect to get lower than tolerating rollbacks of length 40. If we want to do better than this, we need to take another approach known as finalization.

12.9 How to Buy Tickets

Before we look at finalization we will briefly discuss how to buy tickets for the lottery. One approach which does not work is to let people make accounts as they desire and say each account has 1 ticket. If there are 100 honest parties running the protocol and one corrupted party, the corrupted party could simply create 1000 fake accounts and take over the network. This is called a Sybil attack, where the adversary creates multiple identities. We therefore need a solution where a ticket is associated to a resource which cannot be copied or freely created.

12.9.1 Permissions Blockchains

An easy solution is then to verify peoples identity when they create an account. This can be done, for instance, if ten big companies run a blockchain between them: they simply get to have one account each. The assumption on having a majority of honest tickets is then translated into an assumption that a majority of the companies are honest.

12.9.2 Proof-of-Stake

In a fully open peer-to-peer system one cannot control who has an account in the system. The whole idea of the system is that anyone can enter the system. In that case we need some other way to limit how many tickets a given party gets. A popular way to do this when the totally-ordered broadcast is used to implement a cryptocurrency is to make the number of tickets proportional to the holdings of your account. In this case the assumption that most of the tickets are honest translates to the assumption that the owner of most of the money in the system are honest. This makes sense in that it is the totally-ordered broadcast that protects the integrity of the system. Breaking the totally-ordered broadcast would break the cryptocurrency, which would have the money held in the system lose most of it value. Hence the holders of the money in the system have an interest in keeping the system healthy. This principle could be called *You might scratch someone else's Lamborghini, but you are not likely to scratch your own Lamborghini.*

12.9.3 Proof of Work

Another way to fight Sybil attacks is to use proofs of work. This is also sometimes called *one clock-cycle, one ticket.* The idea is that you cannot cheat with how many clock cycles you have, they cannot be copied or freely created. They ultimately require energy to make, so proofs-of-work system could also be said to be build on the principle of *one watt, one ticket.*

Proofs of work systems are not build on the exact form of the system we build above. The above structure was targeted against proof-of-stake systems. In a proof-of-work systems you typically do not win the right to fill the next slot, instead you win the right to extend the tree at a particular node. We will discuss the difference in more detail below. It is, however, instructive to begin with our proof-of-stake skeleton above and then massage it into a proof-of-work protocol.

Recall that the value of your draw is

$$\mathsf{Val}(\mathsf{P}_i, \mathsf{slot}, \mathsf{Draw}) = \mathsf{Tickets}_i \cdot H(\textsc{Lottery}, \mathsf{Seed}, \mathsf{slot}, \mathsf{P}_i, \mathsf{Draw}) \ .$$

If we wanted to put in $\mathsf{Tickets}_i$'s place something proportional to how much computing power you have, what could we put there? The idea is to put there the solution to a hard computational puzzle. The harder a puzzle you can solve, the more computational power you must have.

So what puzzle should we go for? Consider a hashfunction H. It maps into for instance 256-bit strings. The outputs of H on any set of different inputs are believe to look random and independent. So consider $H(c)$ for some arbitrary number c. The probability that the first bit of $H(c)$ is 0 is $\frac{1}{2}$. The same holds for the second bit, etcetera. So, the probability that the first h bits of $H(c)$ are all 0 is 2^{-h}. It can be proven that if you have an experiment which is successful with probability T^{-1} each time you repeat it, then the expected number of times you have to repeat it to see success is T. So, if you pick a number c and start computing $H(c), H(c+1), \dots$, until the first h bits are all 0, then the expected number of times you have to compute H is 2^h. Therefore presenting a number d such that $H(d)$ has h trailing 0's is reasonable proof that you computed H about 2^h times. So to prove that you have a large computational power you could just find and send unto the network a d such that it has many initial 0's.

The above does not completely work, as a malicious party could just copy your d. To prove that *you* did the work, you will hash along your identity, which in this case the the verification key vk_i corresponding to your secret key sk_i. So to prove your computational power you simply compute

$$H(\mathsf{vk}_i, c), H(\mathsf{vk}_i, c+1), \dots$$

and your keep the d which makes $H(\mathsf{vk}_i, d)$ have most initial 0's. Now the proof cannot be copied, it is tied to your identity vk_i.

Before we proceed, let us rephrase the number of tickets carried by $H(\mathsf{vk}_i, d)$ to a simpler term than via the exponential of the number of initial 0. If $H(d)$ is a 256-bit number and has h initial 0's then as a binary number it is about 2^{256-h}. Or equivalently, the number of initial 0's is about $256 - \log_2(H(d))$. So the number of tickets is about $2^{256-\log_2(H(d))} = 2^{256}/H(d)$. So the value of your draw is

$$\mathsf{Val}(\mathsf{P}_i, \mathsf{slot}, \mathsf{Draw}) = (2^{256}/H(\mathsf{vk}_i, d)) \cdot H(\textsc{Lottery}, \mathsf{Seed}, \mathsf{slot}, \mathsf{P}_i, \mathsf{Draw}) \ .$$

The number 2^{256} is the same for all parties, so it does not change who won, it just scales all values. So we might as well set

$$\mathsf{Val}(\mathsf{P}_i, \mathsf{slot}, \mathsf{Draw}) = H(\textsc{Lottery}, \mathsf{Seed}, \mathsf{slot}, \mathsf{P}_i, \mathsf{Draw})/H(\mathsf{vk}_i, d) \ .$$

Proof-of-Burning-the-Planet

This now proposes a number of strategies for how to find the values $H(\mathsf{vk}_i, d)$. We could for instance say that when you enter the network you start computing $H(\mathsf{vk}_i, c), H(\mathsf{vk}_i, c+1), \dots$ and continuously broadcast the smallest $H(\mathsf{vk}_i, d)$ that you ever found. This would give a proof of total amount of energy spent. This would be a proof that you did more than your fair contribution to global warming.

Proof-of-Hardware-Cost

We could also somehow broadcast a new seed Seed every week and then have all parties compute $H(\mathsf{Seed}, \mathsf{vk}_i, c), H(\mathsf{Seed}, \mathsf{vk}_i, c+1), \dots$ for ten minutes and then broadcast the smallest $H(\mathsf{Seed}, \mathsf{vk}_i, d)$ that they found. By having Seed be random and unknown until the competition starts we would ensure that no one could start computing $H(\mathsf{Seed}, \mathsf{vk}_i, c), H(\mathsf{Seed}, \mathsf{vk}_i, c+1), \dots$ before the seed was published. This would mean that the machinery for mining, as it is called, would only be turned on for ten minutes every week. This would burn orders of magnitude less energy. In fact, the main cost in buying tickets would now probably be in hardware cost, as the machines are rarely running. This could make it an advantage to use machines for mining which can also be used for other tasks all the minutes of the week where there is no mining competition.

Towards Bitcoin

A quick look at the equation

$$\mathsf{Val}(\mathsf{P}_i, \mathsf{slot}, \mathsf{Draw}) = H(\textsc{Lottery}, \mathsf{Seed}, \mathsf{slot}, \mathsf{P}_i, \mathsf{Draw})/H(\mathsf{vk}_i, d)$$

makes one ask why we compute the hash function twice, why not just slab all of it into the same evaluation and let

$$\mathsf{Val}(\mathsf{P}_i, \mathsf{slot}, \mathsf{Draw}) = 1/H(\textsc{Lottery}, \mathsf{Seed}, \mathsf{slot}, \mathsf{P}_i, \mathsf{Draw}, \mathsf{vk}_i, d)\ .$$

Well, that works as well in principle, but it gives a system that it very hard to behave rationally in. Note that you have to decide on the slot slot before you start mining. How long into the future should you mine? What Bitcoin did is something similar but dropping the slot number. Instead the slot to be mined was just "the next slot". This was implemented by not hashing a slot number but the block Block that the miner considers the best leaf. The hash that won the lottery is added to Block in the so-called block header. An advantage of this is that the block then at the same time can function as the next seed, as it contains a "random" output of a hash function. It can also be seen that at this point we do not need the draw anymore. The idea of the draw was to tie the lottery to P_i and a given slot. By now we hash vk_i along and we no longer need slots. The value vk_i also replaces P_i. So the value is now

$$\mathsf{Val} = H(\textsc{Lottery}, \mathsf{Block}, \mathsf{vk}_i, d)^{-1}\ .$$

What is described above is still not completely Bitcoin, but it highlights very well how Proof-of-Work were used in Bitcoin: The new blocks Block are fresh and contain a random output of a hash function and can therefore be used as a seed to start a new competition about who has more computational power. The winner of the competition gets to add the next block, and so on.

Notice that the mining competition now start every time a new block Block is produced, as you have to jump to mining on that block. Therefore Bitcoin is basically a Proof-of-Burning-the-Planet system as the mining is going on constantly. This is a huge disadvantage of Bitcoin over for instance proof-of-stake system. There are, however, also some advantages. I you revisit Fig. 12.5 you will see that some slot numbers, like 13, occur in several places. This is because in a proof-of-stake system you win the right to a slot, so you can extend that slot on *all* paths. You are not supposed to do that of course, but a corrupted party could. This does not happen in Bitcoin. If you want to extend at two leafs in the tree, you need to mine at both positions, which would have you split your mining power. It is therefore rational to mine only at the best leaf in Bitcoin. In particular, if we would make a tree similar to Fig. 12.5, all corrupted nodes would appear in only one position. It would therefore be much easier to get enough superslots to match the filler slots, and hence one would get much better parameters in the security analysis.

This unfortunately means that there still is a real choice between proof-of-work systems and proof-of-stake systems. The first is more secure, but is burning the planet. The later are much, much more energy efficient, but less secure. It is an open research question to find a lottery system which is as energy efficient as proof-of-stake and as secure as proof-of-work.

Exercise 12.2 (static proof-of-stake) Start from your solution in Exercise 6.13 and use parts of your code from Exercise 11.1. The code in Exercise 6.13 should already be a distributed ledger with authenticated transactions. However, it does not have total order. Change it such that it gives a total order of all transactions and rejects transactions that would bring an account into minus. Do it by adding a proof-of-stake, tree-based, totally-ordered broadcast. Implement total-order using a tree-based block-chain as the one in Section 12.7 based on proof-of-stake as described in Section 12.9.2. You should not implement finalization or dynamic parameters. In a bit more detail, implement it as follows:

1. The initial seed Seed is picked by you and hardcoded into the genesis block. Note that this seed does not change, as you do not have to do dynamic parameters. All the signatures in the lottery uses this seed forever.
2. Transactions are conducted in the unit AU.
3. The genesis block contains ten special public keys which by definition have 10^6 AUs on them. All other accounts have 0 AU on them initially. These ten accounts are generated by you, and you know the secret keys.
4. Transactions are in positive, integral AUs.
5. A transaction must send at least 1 AU to be valid.
6. A block can contain any number of transactions, and might even contain as little as no transactions if there are none to add to the block.
7. SlotLength is 1 second. You might set it larger if your signatures are very slow to compute. Recall that you need to compute one signature per slot.
8. To take part in the lottery and making blocks you need an account in the ledger with a positive balance. Your number of tickets is the balance of the account. Throughout the system your number of tickets is the balance in the genesis block, so only the ten accounts you created can be part of running the system.
9. The signature keys used in the lottery is the same as those used in the ledger system.
10. Make the system run with 10 peers.
11. Set the hardness such that your system creates a new block about every 10 seconds.
12. A block is not added to the tree unless all transactions are correctly signed and valid (they make no account go below 0 at any point).
13. When a transaction is made, the receiver gets 1 AU less than what was sent. This is a transaction fee.
14. When a new block is made, then the account of the block creator gets 10 AU plus one AU for each transaction in the block.

Your report should include:

1. How can the TA run your code.
2. How did you test your code. Remember to test also against some of the peers being malicious.
3. How did you test that agreement was achieved.
4. When the system is not under attack, how many transactions per second can the system handled. A transaction is not counted as done until it has been ordered and the balance of the accounts have been updated with that transaction.

13

Finality (2018)

Contents

13.1 The Finality Problem

Agree. I know that we agree. I know that we all know that we agree. ...

13.2 Finality Layers: Pruning Ghost Branches

We will now briefly discuss how finality can be added. Recall that if you want any real guarantee that a block will never roll back when the system is under attack, you have to look very far back into the tree. However, when the system is not under attack, the part of the tree where parties are not in agreement is very short, maybe just a few blocks. This asks the question of whether we can do better than just waiting for the possible worst case to be gone and instead detect when a block is final? We can! We now sketch how it is done on our Toy Chain blockchain.

13.2.1 The Finalization Committee

There will be a finalisation committee $\mathsf{FinalityCommittee}$, which is just a set of verification keys vk_i. This could for example consists of servers that we have seen are able to stay online stably for a long time. We call the members of the committee the finalisers. Each vk_i has a number of votes, denoted by Votes_i. This could for instance just be their number of tickets in the tree layer. It can also be the sum of tickets of parties from the tree layer that delegated their voting power to the voter. What is important is that the finalisation committee is mostly honest. Specifically we assume that at most a small fraction of the votes are corrupted. For a set S of finalizers let

$$\mathsf{Votes}_S = \sum_{i \in S} \mathsf{Votes}_i \ .$$

Let $\mathsf{Corrupt}$ be the set of corrupted finalisers. What we require is that

$$\mathsf{Votes}_{\mathsf{Corrupt}} < \mathsf{Votes}_{\mathsf{FinalityCommittee}}/5 \ .$$

We assume that we have a weak multi-valued BA WMVBA. Recall that this is a protocol where all parties input some bit string and will agree on some bit strong or $\perp$. If all honest input the same bit string, then that will be the value agreed on. The WMVBA should be secure as long as $\mathsf{Votes}_{\mathsf{Corrupt}} < \mathsf{Votes}_{\mathsf{FinalityCommittee}}/5$. Such a protocol can be made by massaging the protocol in Section **??**. Simply replace n by $\mathsf{Votes}_{\mathsf{FinalityCommittee}}$ in the protocol and always count votes of the parties sending the values instead of just the number of parties.

13.2.2 Common Prefix

Before we look at finalization we introduce the $\mathsf{common\ prefix}$ property. It is a property closely related to limited roll-back. It says that if you take any two nodes in the honest tree and walk back c steps on both of them, then the two corresponding positions are on a common chain. Another way of defining it is to say that there is a main chain, called the stem, such that no branch from the stem has length more that c. Consider Tree^7 in Fig. 12.2. It has $\mathsf{CommonPrefix} = 2$. Namely, if you start from nodes 15 and 18 and walk back two steps you are on positions 8 and 11, and clearly 8 and 11 are on the same path. A low $\mathsf{CommonPrefix}$ means that the tree is currently not branching too much. It turns out that low $\mathsf{CommonPrefix}$ is very closely related to limited roll back.

Exercise 13.1 (Common Prefix versus Limited Rollback)
Show that if a tree has common prefix $\mathsf{CommonPrefix}$, then while the honest tree does not change there cannot be a rollback of length more than $\mathsf{CommonPrefix}$.

13.2.3 One Finalization

Our idea for finalization is the following: We will ask the honest parties when they are at some agreed height L to look back Δ steps in the tree. Note that if $\Delta \geq \mathsf{CommonPrefix}$ they now look at nodes on the same path. Since they look at the same height, this means that they actually look at the same block. So if we run WMVBA on the block they look at, the WMVBA will actually agree on that

```
1. It is required that $\Delta$ is a integer and that $\Delta \geq 1$.
2. Wait until $\text{Len}(\text{BestPath}(\mathsf{Tree})) \geq h + \Delta$.
3. Let $\mathsf{Block}_i = \text{BestPath}(\mathsf{Tree})[h]$.
4. Let $baid = (fid, \Delta)$ and run $\mathsf{Block} = \mathsf{WMVBA}(baid, \mathsf{Block}_i)$.
5. If $\mathsf{Block} = \perp$, then let $\Delta = 2\Delta$ and go to Step 2.
6. If $\mathsf{Block} \neq \perp$, then wait until $\mathsf{Block}$ occurs in the tree at height $h$ in some branch
   and then output $(\mathsf{Block}, \Delta)$.
```

Figure 13.1 The protocol $\mathsf{AgreeOnBlock}(fid, c, h, \Delta)$ agrees on the block at height h. The input fid is a unique identifier for this run of the protocol. Input c is a counter of previous finalizations. Input h is the height of the block to finalize. Input Δ is a "delay" in when to start the finalization.

block. If $\Delta <$ CommonPrefix, then WMVBA might result in $\perp$, but then we just try again, this time with a higher Δ.

Now consider the protocol in Fig. 13.1 which waits until the block at height h in the tree can be agreed upon. We first show that the protocol eventually terminates. Then we discuss what the output means and how it is used.

All honest parties will start running the protocol as their trees eventually reach height h by the tree growth property. Therefore they will all start running $\mathsf{Block} = \mathsf{WMVBA}(\mathsf{Block}_i)$. If $\mathsf{Block} \neq \perp$, then some honest party gave Block as input. Therefore it will eventually appear at height h in the tree of all honest parties in some branch. At this point all honest parties terminate. If $\mathsf{Block} = \perp$, then they run $\mathsf{Block} = \mathsf{WMVBA}(\mathsf{Block}_i)$ again but only after reaching height $h + 2\Delta$. After f failures to agree they run $\mathsf{Block} = \mathsf{WMVBA}(\mathsf{Block}_i)$ at height at least $h + 2^f\Delta$.

Let P be the common-prefix value guaranteed by the limited rollback property, i.e., the number of block the honest parties have to go back to be on a common path. When $2^f \geq P$, then all honest parties input the same $\mathsf{Block}_i = \mathsf{Block}$ to WMVBA as they are on a common prefix and look at the same height h. Hence the output will be Block by the pre-agreement validity property. At this time the protocol terminates. Since 2^f grows very quickly we will very quickly find a Δ which gives agreement.

13.2.4 Continuous Finalization

The above section described how to do one finalization. We would like to continuously finalize block in the tree as it grows. The question is then when to try to agree again, i.e., how do we pick he next h and how do with pick the next Δ. For starters we could agree on the next block after the genesis block ($h = 1$) and be optimistic and use $\Delta = 2$. When the agreement on $h = 1$ terminates we need to pick the next h and Δ. When we pick h, we do not want to attempt agreement too far into the past as it will then take too long to agree on the current block. Say the agreement on $h = 1$ terminates when we are at height 1000. Then we do not want to start agreeing on block 2. We should probably go for $h = 1000$. Once block number 1000 is agreed on, so are all the block on the part from genesis to the agreed block. Similarly, we do not want to agree too far into the future as it will then also take too long to agree on the current block. If we are now at block 1000 we do not want to pick $h = 2000$ as it will be a while until we start the attempt to find agreement. We therefore want to pick h close to the current block. And if we are slightly off, we want to pick h in the past, as this will allow us to start the finalisation protocol immediately. We can, however, not set h to be "the current block" as the parties might then have different views on what h is and it is important for AgreeOnBlock to work correctly that they all agree on h. Note, that when AgreeOnBlock terminates with output Δ, then the successful run was started when the parties were around height $h + \Delta$. So it is a fair guess that when AgreeOnBlock terminates, then all correct processes are at height about $h + \Delta$, maybe a bit higher. We can therefore use $h' = h + \Delta$ as the next block to try to agree on. This is the rational behind the protocol in Fig. 13.2.

---

1. Let $\Delta = 1$. Let $h' = 0$. Let $h = 1$. Let Agreed denote the height and value of the latest agreed block. Initially Agreed $= (0, \mathsf{Block}_0)$, where Block_0 is the genesis block. Let Agreeds $= \{\mathsf{Agreed}\}$.
2. Wait until BestLeaf(Tree).LastAgreed $= h'$. [a]
3. Run $(\mathsf{Block}_h, \Delta_h) = \mathsf{AgreeOnBlock}(fid, h', h, \Delta)$.
4. Let Agreed $= (h, \mathsf{Block}_h)$. Let Agreeds $=$ Agreeds $\cup \{\mathsf{Agreed}\}$.
5. Let $h' = h$. Let $h = h + \Delta_h$.
6. If $\Delta_h > 2$, then let $\Delta = \Delta_h/2$. Otherwise let $\Delta = 1$.
7. Let $c = c + 1$ and go to Step 2.

[a] On a first read, you can ignore this step. We will return to explaining it later. On a first read, pretend the line is not there.

Figure 13.2 The protocol KeepAgreeing(fid) continuously agree on higher and higher blocks.

We will later see that it is important not to try to agree on the next block before the previous finalization terminated.

Note the when AgreeOnBlock terminates with output Block $\neq \perp$ and Δ, then Δ was the lowest value that allowed to achieve finalization. It would therefore make good sense to do the next run with the same Δ as input. The reason why we use $\Delta = \Delta_h/2$ is that if for a short while the network is an bad conditions the tree layer will tend for fork a lot and Δ will grow to accommodate this. By using $\Delta = \Delta_h/2$ the value of Δ will find a lower value again when the network is in good conditions. In practice one might want to lower the value in a slower manner to avoid too much flickering.

13.2.5 How do We Make a Final Block Final?

Recall that a block is final only if it can never happen that there will be a rollback behind that block. So far we are just updating the variable Agreed, which of course will not affect what the tree layer is doing. It might even be that at the time that a block is stored into Agreed, the tree layer did a rollback and hence there is not a single honest party on the path the contains Agreed.Block. We now somehow have to ensure that all honest parties get on the path that contains Agreed.Block and that they never do a rollback behind Agreed.Block.

Our plan is simple. We will update the PathWeight rule such that all parties prefer the path with most finalized blocks. We do that by hacking the rule to prefer paths with more finalized blocks. To do that we need the number of finalized blocks to be a function of the path, right now it is defined via both the path and Agreeds. To make it a function of just the path, we will simply ask block producers to include in each new block a pointer to what they consider the previously last finalized block. To avoid malicious parties lying we will not accept a block until we verify that the block pointed to is final. This is done via the set Agreeds.

Finalization Meta-Data: When a new block Block is made, then we include in the meta data M a new field LastAgreed. It is the maximal height in the path from genesis to Block where a block from Agreeds is sitting. This is done via GetMetaData().

When a block Block is received then it is not considered valid until PathTo(Block)[Block.M.LastAgreed] $\in$ Agreeds. Furthermore, Block.M.LastAgreed should be larger than LastAgreed on all nodes in PathTo(Block). This is check is done by ValidMetaData.

Prefer-Agreed-Blocks Rule: We update the PathWeight rule such that the number of finalized blocks counts first. Specifically, for any PathWeight, let $\text{PathWeight}^{\text{FIN}}$ be the ordering which first sorts on the height of the highest finalized block and then sorts like PathWeight. We use that rule in the tree layer when we want finalization.

13.2.6 What is Finalization?

We now want to prove that the above protocol gives us finalization. But let us first figure out what to prove. Recall that now each block Block points to a block that it "considers" the last finalized block. We can then define a block Block in the honest tree to be a finalized block if there is another block Block′ in the honest tree which points to Block as begin the last finalized block from the point of view of Block′. A bit more precise:

Definition 13.1 (final blocks) Let HonestTree be the honest tree. Let Agreeds be the union of the sets Agreeds_i held by honest parties. We define a block Block $\in$ HonestTree to be a finalized block if Block $\in$ Agreeds.

△

Definition 13.2 (tree final blocks) Let HonestTree be the honest tree. We define a block Block $\in$ HonestTree to be a tree finalized block if there exists Block′ $\in$

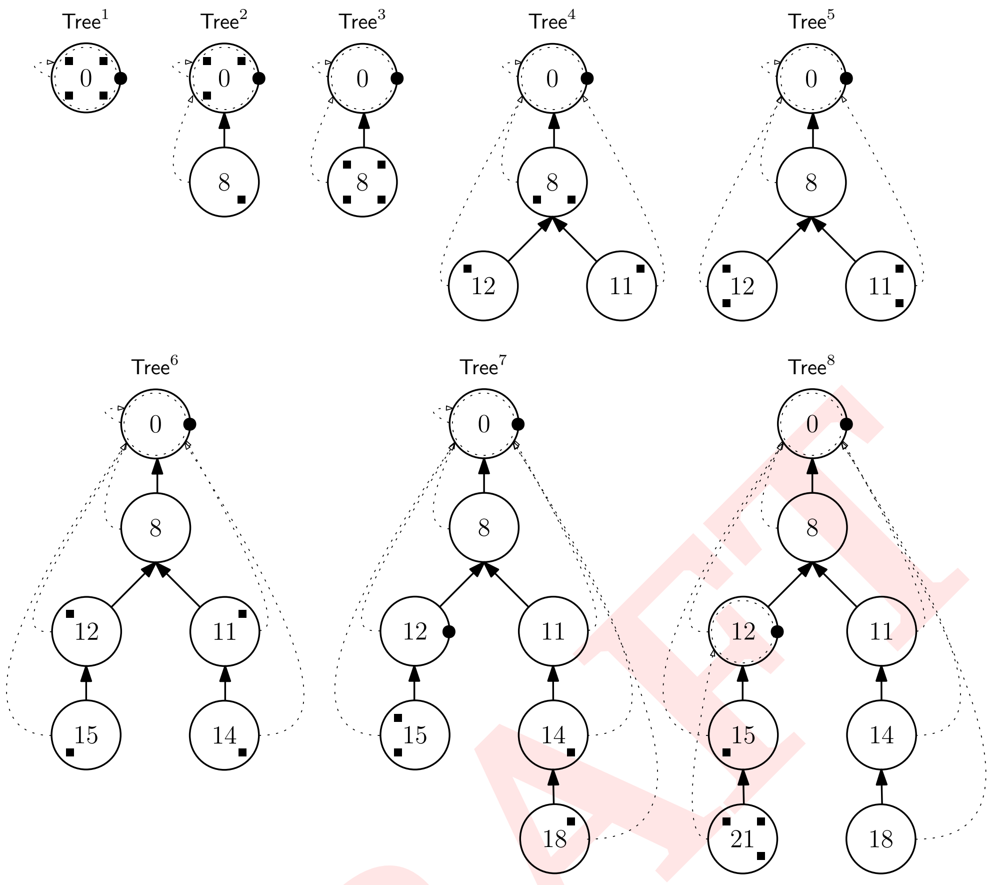


Figure 13.3 We depict as in Fig. 12.2. If we put a dotted circle inside a block it means that it is a tree final block. From the beginning only genesis is tree final. If we put a fat dot on the border of a block it means that the block is finalized, i.e., it is in Agreeds. The dotted arrows are the LastAgreed pointers that in each block point to the last final block further up the tree. From Tree^1 to Tree^7 the execution is as in Fig. 12.2. While this is going on the finalization protocol is running in the background with the very agressive setting $\Delta = 1$ and with $h = 3$. Therefore as all parties reach height 3 in Tree^7 (blocks 15 and 14 are both at height 3) they run a WMVBA. Two honest parties vote for 12 and two honest vote for 11. So the possible outcomes are $11, 12, \perp$. Say the output is 12. Now 12 is added to Agreeds. This is indicated by the fat dot. Note that 12 is not a tree final block yet! But now one of the parties that have 12 in Agreeds wins slot 21. It then adds block 21 with a LastAgreed that points to block 12. Now block 12 is considered tree final. Note that that he best chain rule says $(0, 8, 12, 15, 21)$ is longer than $(0, 8, 11, 14, 18)$, as $(0, 8, 12, 15, 21)$ has two final blocks whereas $(0, 8, 11, 14, 18)$ only has one final block. Therefore the parties on blocks 14 and 18 will eventually jump to the chain $(0, 8, 12, 15, 21)$. This is illustrated in Tree^8.

HonestTree such that $\mathrm{PathTo}(\mathsf{Block}')[\mathsf{Block}'.M.\mathsf{LastAgreed}] = \mathsf{Block}$. We let $\mathrm{FinalBlocks}(\mathsf{HonestTree})$ be the set of final blocks in HonestTree.

△

Notice that the finalization layer considers a block final when it is in Agreeds, but it does not try to agree on the next finalized block has become tree final. This is important for security. It guarantees that anyone that runs the next WMVBA is below the previous finalized block in the tree. An execution is depicted in Fig. 13.3.

13.2.7 Arguing Finalization

Consider now our blockchain above but with finalization meta-data added and the prefer-agreed-blocks rule. We want to argue that all block in Agreeds lie on a single path, namely PathTo(Agreed). This is clearly the case in the beginning, where only genesis is in Agreeds. So we can argue inductively. So assume we are in any situation where $\mathsf{Agreeds} \subseteq \mathrm{PathTo}(\mathsf{Agreed})$. By the check

$$\mathrm{BestLeaf}(\mathsf{Tree}).\mathsf{LastAgreed} = h'$$

in Step 2 in KeepAgreeing, no party will run the next AgreeOnBlock until they are on a path containing Agreed, as Agreed is the unique block in Agreeds with $\mathsf{Agreed}.h = h'$.

As an illustration consider Fig. 13.3. After the parties ran the WMVBA that put 12 in Agreeds, none of them will start the next WMVBA until they are on a path with a leak that points to 12 as LastAgreed. So, they do not run the next WMVBA until they are block 21 or below.

Consider now the period of time until the next run of AgreeOnBlock terminates at some correct party. In this period all correct parties on the path containing Agreed will stay on a path containing Agreed as no block with higher $\mathsf{Agreed}.h$ will be added to Agreeds at any correct party: that would namely require the next run of AgreeOnBlock to terminate. So in that period all parties that start running AgreeOnBlock will stay on a path containing Agreed until the first correct party terminates AgreeOnBlock. Therefore the block returned by AgreeOnBlock will clearly be a block which sits on a path containing Agreed. Hence the induction invariant is maintained.

As an illustration consider Fig. 13.3. One thing that could go wrong there is that the next final block is 11, 14, or 18. But we already argued that the honest parties do not run the next WMVBA until in block 21. So they will all vote on a block that sits below 12. Therefore the outcome cannot be 11, 14, or 18, as a WMVBA only outputs d if at least one honest parties input d.

13.2.8 Pruning Ghost Branches

Note that when we analysed the tree layer, one of our big problems were ghost branches, i.e., branches made up of corrupted slot winner that have withheld their block and build a long chain in secret. An advantage of finalization is that a ghost branch that sit lower than a finalized block can never be used again. It is namely easy to see that the finalization layer only will select a block that was in the honest tree when the finalization terminated. This is because WMVBA only selects values input by an honest process. So to make a ghost block a candidate for Agreed.Block the corrupted parties must send it to the honest parties. But then it is not a ghost anymore! And if a block $\mathsf{Block}^{\text{GHOST}}$ is not a candidate to win, some other block Block will win and hence the honest processes will never switch to $\mathsf{Block}^{\text{GHOST}}$: the ghost branch has been pruned. This allows a much better analysis when there is rapid finalization. We will not dive into this here.

13.3 The CAP Theorem

We conclude the chapter by discussing the so-called CAP Theorem. It relates three important concepts, consistency, availability and partitioning.

A network is partitioned if there are two parts of it that cannot talk to each other. More precisely, a partitioning from time t_1 to t_2 of a network with participants $\mathcal{P}$ is a partitioning of $\mathcal{P}$ into two disjoint sets $\mathcal{P}_1$ and $\mathcal{P}_2$ such that

- $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2$
- Between t_1 and t_2 no message is sent from a party in $\mathcal{P}_1$ to a party $\mathcal{P}_2$ or from at party in $\mathcal{P}_2$ to a party $\mathcal{P}_1$.

A partitioning could be because an important cable is cut or just because messages are slow in a period.

Consistency of a system can mean various things, in general any safety property. In totally ordered broadcast it for instance means that all parties see all messages in the same order.

Availability talks about a system delivering recent values, or in general just being operational in a timely manner.

The CAP theorem then roughly says that during a partition, the system has to choose between availability and consistency.

To make it more concrete, think about a world wide system for ATMs keeping track of how much money people have on their accounts and for paying out the money in currency if there is enough on the account. There is a user U with 100 on her account. Safety would be that if she tries to withdraw the 100 twice, then the system will still only hand out at most 100. No account is allowed to become negative. Availability would be that if she tries to withdraw her 100 in any ATM, then they will actually be dispensed.

To illustrate the CAP theorem, assume now that the internet partitions so there is not communication between Europe and America for an hour. Imagine the following three scenarios.

E During the partition hour U walks up to an ATM in Europe and withdraws 100.

A During the partition hour U walks up to an ATM in America and withdraws 100.

EA During the partition hour U and an accomplish withdraw 100 from her account in Europe and in America at the same time.

If during the partition the system is available in Europe, then in scenario E it will pay out the funds via the European ATM. Since there is no way for the European ATM to know whether it is in scenario E or EA during the partition, it means that if during the partition the system is available in Europe, then in scenario EA it will pay out the funds via the European ATM. Similarly, if during the partition the system is available in America, then in scenario EA it would pay out the funds via the American ATM. But then safety is broken in EA: U walks away with 200 in cash. So any system, no matter how it is implemented will during a partition as above have to give up on availability in Europe or America, or give up on safety.

Best of both Worlds

Just because you have to pick between consistency and availability does not mean that you have to do it at a system level. If you are smart and took a course on distributed systems, you can make it a dynamic choice of the user of the system. We illustrate this using our blockchain from above with finalization to implement the ATM system.

To help the reader better understand the blockchains behaviour during a partition, an example of a partitioning is depicted in Fig. 13.4.

When a users U withdraws an amount m, the ATM will broadcast it on the blockchain. If the users puts money on the card, the ATM broadcast that too. At any node in the tree the holdings of the account is just the sum of withdrawals and deposits up to that node. Any withdrawal that would bring the account into the negative is ignored.

Assume that a withdrawal is an amount signed by the card of the user. The signature is given to the ATM that broadcasts it to the blockchain. There are now three places where the ATM could choose to pay out the cash.

1. When the ATM gets the signature.
2. When the ATM sees the signature appear in the tree.
3. When the ATM sees the signature appear in the tree and it is in a node that is final.

The first approach might break safety even when there is no partition. In scenario EA the user would get 200 out in cash. But for tiny amounts paid out to a known customers it might be OK. The user can only steal a small amount, and we could send the legal system after her afterwards.

The second approach guarantees availability during a partition, but not safety: The tree would continue to grow both in Europe and in America. But they would grow independently of each other: the honest tree would develop two long branches after t_1. But there would still be throughput. In particular in both E

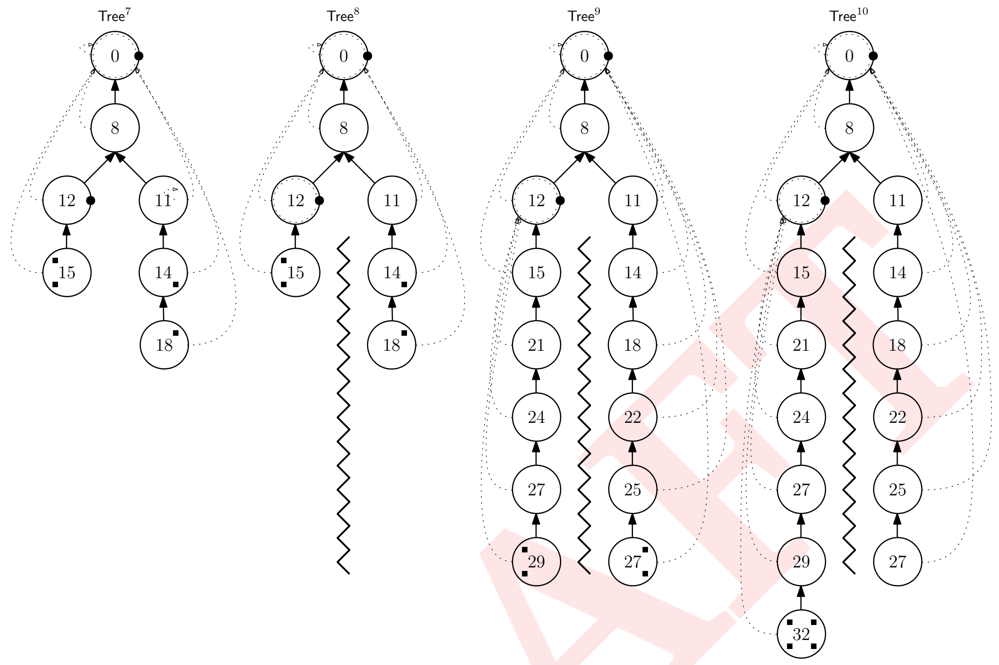


Figure 13.4 We depict as in Fig. 12.2. The execution runs as Fig. 13.3 until Tree^7. Then exactly when the WMVBA terminates, the network partitions with the two honest parties in block 15 on one side and the other two honest parties on the other side. Only the parties in 15 learn the result of the WMVBA. Now the parties on the left considers 12 to be the last final, whereas the parties on the right consider 0 to be the last final. During the partition there are not enough parties on either side for the next WMVBA to terminate: it has deadlocked. Therefore LastAgreed is not updated on either side during partition: finalization has shut off. When the network joins again, the parties on the right receive the result of the first WMVBA, receive block 21, learn that 12 is final, and jump to the left chain as it has more finalized blocks. Soon finalization turns back on as there are now enough parties for the second WMVBA to come out of the deadlock.

and A the transaction would eventually appear in the branch and the ATM would pay out. So clearly safety is lost in EA.

The third approach guarantees safety, but not availability. Recall that WMVBA always outputs the same to all parties. And what it outputs is a block from the tree in some branch. In this case, either from the European branch or the American branch. It is clear that if it outputs a block from the European branch, then it does not terminate in America until the partition is gone. Namely, the block from the European branch is a new value created after the partition, so there is no way to communicate it to America. Hence WMVBA of course can't output it in America either. Similarly in the other direction. So during a partition finalization will stop, and hence the ATM cannot pay out the money. It might be that by chance there are so many finalizers on one side of the partition that WMVBA for instance finalization can keep running in Europe. Then the European ATM can pay out, but the Amerian one cannot. But for a random partitioning, most like finalization will stop on both sides.

Note that the fact that finalization stops is a feature! It prevents double spending in the EA scenario. And the CAP theorem tells us that when the system has safety, then it cannot have availability during a partition, so finalization must stop in either Europe or America.

But note that importantly, the ATM can pick its own strategy. If it is paying out a small amount to a known customer, it could just do it in good faith. If instead the card is being used to pay a Lamborghini in a car shop to a shady looking guy with a foreign card, probably you want to wait for finalization before handing over the keys.

14

Network Security Mechanisms (DRAFT)

Contents

14.1 Authenticated Key Exchange

In several of the previous chapters, we have allowed ourselves to assume that secure channels can be established between pairs of entities in our system, where secure means that en external adversary cannot see what is sent on the channel nor can he modify what is sent without being detected.

In this chapter we cover techniques for establishing such channels based on an insecure network such as the Internet. Thus our threat model throughout will be that the adversary can see and modify all communication as he likes. Also, entities you communicate with may be adversarial/Byzantine, we assume nothing about how many.

We will assume that two entities A, B wanting establish a secure connection both have certificates containing public keys pk_A, pk_B, see Chapter 10 on Key Management for details on certificates and how to manage them.

The idea is now to use the public key pairs in some appropriate protocol to establish a short-lived session key K, which is then used to provide secrecy and authentication by secret-key techniques. Note that this is an example of the combination of long-lived keys and session keys that we also discussed in Chapter 10 on key mangement. Another strong motivation is efficiency: we do not want to encrypt large amounts of data using a slow public-key algorithm.

It must be ensured, however, that both parties can be sure that they agree on K, and that they share K with the right entity, i.e., if A thinks he has been talking to B and vice versa, it should be ensured that K is really known only to A and B.

Here is a more precise definition: An *authenticated key exchange protocol* is a protocol for two parties A, B. Each party starts the protocol with the intention of establishing a key with some other party. At the end, each party outputs either "reject", or "accept" as well as a key. The protocol is said to be secure if the following three conditions hold:

Agreement Assume that A intends to talk to B and B intends to talk to A. Assume also that both parties accept and A outputs key K_A, while B outputs key K_B Then $K_A = K_B$.

Secrecy and Authentication Assume A intends to talk to B, and accepts. Then it must be the case that B participated in the protocol and if B also accepts, he did indeed intend to talk to A. Furthermore, the adversary does not know the key K that A outputs. A symmetric condition holds for B.

Freshness if $A(B)$ follows the protocol and accepts, it is guaranteed that the key K that is output is a fresh key, i.e., it has been randomly chosen for this instance of the protocol, independently of anything else.

At first sight, it may seem that this definition is overly complicated. Why don't we just demand that both parties always agree on whether the protocol was successful? However, this is too much to hope for: We have to remember that we assume the network we use for the protocol is not secure and the adversary can do whatever he wants to the communication. So if, for instance, A is supposed to receive the last message in the protocol, the adversary can always stop this message or replace it with an incorrect one. Then B thinks everything is fine, but A concludes that it failed.

Note also that it is easy to come up with simplistic protocols that may seem fine but do not satisfy this definition. For instance, suppose A just sends a session key to B encrypted under B's public key. This would not satisfy the Secrecy and Authentication condition: B cannot verify that the encrypted key comes from A, and A cannot verify that B is out there to receive it.

Finally note that the freshness property guarantees that an adversary cannot make you reuse an old session key, which is an undesirable thing to do as we discussed in the key management chapter.

In a nutshell, one can say that an authenticated key exchange protocol should create a situation that is equivalent to having a totally trusted key distribution center, that sends privately a fresh and secret key to both parties – then the worst an adversary can do is to stop the communication.

14.2 How to not do it

Experience shows that authenticated key exchange is harder to achieve than it may seem. As an example, consider the following protocol, proposed in 1978 by Needham and Schroeder. We assume in the description that $E_{pk}()$ is an secure encryption scheme as discussed in the Chapter 5, and that A and B already know each other's public keys:

1. A chooses nonce n_A and sends $E_{pk_B}(ID_A, n_A)$ to B.
2. B decrypts, checks ID_A, chooses nonce n_B and sends $E_{pk_A}(n_A, n_B)$ to A.
3. A decrypts, checks that the correct value of n_A appears in the result, and sends $E_{pk_B}(n_B)$ to B.
4. B decrypts and checks that the correct value of n_B appears in the result.
5. If the checks executed are OK, each party computes the session key as some fixed function of n_A, n_B.

Recall that *nonces* are values chosen randomly from a large domain, so that we may assume they will repeat with negligible probability.

It may seem very plausible if both A and B end up accepting the values they see, we have essentially what we wanted above, namely A, B agree on the values of n_A, n_B, and no one else knows these values. After all, only B can decrypt values encrypted with pk_B, and similarly for A. So it should be secure to generate the session key from the nonces.

Unfortunately, this turns out to be false! It is possible for a third user E with a certified public key to mix two instances of the protocol, and fool B. This goes as follows:

1. Suppose A starts a session with E, thus A will send $E_{pk_E}(ID_A, n_A)$ to E.
2. E decrypts this and starts up a session with B, pretending to be A. So E will send $E_{pk_B}(ID_A, n_A)$ to B.
3. B decrypts this, finds the right ID in the result, and sends $E_{pk_A}(n_A, n_B)$ to E.
4. E is of course not able to decrypt this, but can instead simply forward the message $E_{pk_A}(n_A, n_B)$ to A.
5. A will decrypt this, find a result that has exactly the form he expected, and will therefore return $E_{pk_E}(n_B)$ to E.
6. E can decrypt, find n_B, and is now able to send $E_{pk_B}(n_B)$ to B.
7. When B decrypts, he will accept since he finds the right value of n_B in the result.

As a result, if we went on to use this as a basis for creating a session key, B would believe he was communicating securely with A, but is in fact talking to E, and this of course completely defeats the purpose of the protocol.

However, the intuition that suggests that the protocol should be secure is not completely unreasonable: If only one instance of the protocol is running at any given time, no attacks are known. Indeed, the attack we saw is crucially based on the idea that several instances of the protocol are running at the same time, and the adversary is able to "mix" these instances together in an unfortunate way.

Finally note that the in the attack above the user A actually initiates a protocol with an evil party E. One might ask, why would A do that? However, think of how many servers you connect to, daily and willingly (e.g., a website you visit) or unwillingly (e.g., a website serving ads as part of a website you actually wanted to visit). It is unrealistic to assume that everyone you connect to is honest and incorruptible, and therefore any realistic threat model for authenticated key exchange must include the scenario in which a user connects to a maliciously controlled, or Byzantine, server.

14.3 The Secure Socket Layer Protocol

One of most common solutions for authenticated key-exchange used today, is the Secure Socket Layer protocol (SSL). It takes a different approach than Needham-Schroeder by using digital signatures in addition to the encryption. The basic idea being that one party must sign a nonce chosen by the other. This does not automatically solve the type of problem we have just seen, however, the latest version of SSL does appear to be secure against this type of attack. SSL has effectively been replaced by Transaction Layer Security (TLS), but there are only very minor differences, and the protocol is often referred as SSL anyway.

To understand how SSL fits into Internet traffic, we need to recall some information on how Internet traffic works: first of all, Internet traffic is split in small packages using the IP protocol, where each packet has a sender and receiver IP address. Furthermore, the TCP protocol which is used on top of this requires further structure on packets. The TCP protocol is connection oriented, which means that particular types of packets are sent and acknowledged in order to open a connection. Later packages can be recognized as belonging to a connection established earlier, and TCP has a built-in mechanism ensuring that packets can be placed at the receiver in the order they were sent, and that a resend will be requested if a packet is missing.

In practice, SSL is used to set up secure http-connections on the Internet. This means that in the standard protocol stack, SSL is placed between the application and the TCP/IP transport layers. More concretely, the data that SSL transports are put inside TCP/IP network packets, so these packets can be moved to the right place by the TCP/IP connection, but the contents of the packets can only be handled by SSL compliant clients or servers.

SSL is always executed between a Server and Client, and in its basic form, it requires that both Server and Client have public-key certificates and access to the corresponding private keys. In addition, both must be able to verify each others' certificates, i.e., a certificate chain must exist, connecting the Server's certificate to a public key known to the Client a priori, and vice versa. It is also possible, however, to do a "one-sided" SSL, where only the server has a certificate. In this case, the Client can verify the identity of the server it is talking to, but the Server cannot verify the identity of the Client. This is in fact the most common use case and we mention later what we can achieve in this case.

SSL is composed of several protocols:

Record Protocol which is responsible for the "raw" transmission of data. The other protocols below rely on this one for transporting data. It uses a so called *cipher spec* to determine how to encrypt/decrypt and authenticate/verify data. The cipher spec says which encryption and other algorithms to use for this. In particular, the spec may be empty, i.e., specify that no encryption should be done. The typical case, however, is that encryption and authentication algorithms are specified, and client and server store keys for both purposes. The record protocol processes a packet by first adding a sequence number and a message authentication code to the raw data, and then this entire string is encrypted. The receiver expects to receive packets in numeric order and halts if this does not happen. This prevents replay and works because we rely on TCP/IP to make sure that packets will (usually) arrive in the order they were sent.

Handshake Protocol This is the part of SSL that does authenticated key exchange. Using SSL terminology, the job of the handshake protocol is to bring server and client from a state where they have an empty cipher-spec and no common keys, to a point where they have negotiated which algorithms to use, and have done an authenticated key exchange. The client initially sends to the server a list of the cryptographic algorithms it supports, in decreasing order of preference. The server then in the next message says which of the possibilities it has selected.

Then the actual authenticated key exchange is done, this will be discussed in more detail later.

Finally, the parties exchange Change Cipher Spec messages (see below) to signal that we should now start using the negotiated algorithms and keys.

Change Cipher Spec Protocol This is just a one message protocol where one party tells the other to change from one cipher spec to another that has just been negotiated.

Alert Protocol This is a one message protocol used to signal error messages to the other side.

It is important to understand that even if we verify theoretically the security of the main part, namely the actual key exchange protocol, this does not guarantee security of the entire SSL construction. For instance, since the set of crypto-algorithms used is not fixed but is negotiated, an adversary may try to manipulate messages between honest players to try to make them use an algorithm for encryption that is weaker then what they can actually use. Note that this may be possible since, while we are doing the handshake, the communication is not yet protected. Indeed, there has been analysis of earlier drafts of SSL pointing out weaknesses of this form. Later versions of SSL fix this problem by including the list of algorithms in the data that is authenticated in the final step of the handshake.

Another issue is that we should of course do a Change Cipherspec just after the Handshake, so we can start using the session keys we just established. The first version of SSL 3.0 did not explicitly demand this, and as a result some Internet servers did not check that a Change Cipherspec was done after every handshake. Hence an attacker could just block the Change Cipherspec message from the client, and then the following communication would be in the clear! This has now been fixed, but serves as a good example of how hard it is to describe standards in a fool-proof way.

Fortunately, the community has learned from these mistakes and hence for the latest version, TLS 1.3, the design has involved academic cryptographic community, from the beginning, and so this newest version has formally proved security guarantees?

14.3.1 The SSL key exchange

We give here a description of the main part of SSL, the actual key exchange protocol. It is important to note that we have stripped away many details from the actual SSL specification, in order to focus on the essential concepts. Also there are several variants of the handshake, and we only describe one of them here.

1. C sends a hello message containing nonce n_C.
2. S sends a nonce n_S and its certificate $Cert_S$ (containing the public key pk_S of S).
3. C verifies $Cert_S$, and chooses a so called pre master secret pms at random. C sends $E_{pk_S}(pms)$ to S, also C sends its certificate $Cert_C$ to S, plus its signature sig_C on the concatenation of n_C, n_S and $E_{pk_S}(pms)$.
4. S verifies $Cert_C$ and sig_C. If OK, it decrypts pms.
5. S sends to C a "finished" message containing essentially a MAC on all messages he has sent and received sent in this instance of the protocol, with pms as the secret key.
6. C verifies the MAC, and if OK returns its own finished message to S, also with a MAC on all messages sent and received up to this step (note that this now includes the finished message from S, so one cannot just repeat the previous message). S verifies the MAC when it is received.
7. At this point both parties derive from n_S, n_C and pms a set of keys for secret-key authentication and encryption of the following data exchange. Typically this is done using a hash function.

The idea here is that S authenticates itself by being able to send a finished message with a correct MAC, thus proving that it was able to compute pms and hence knows sk_S. The client authenticates itself by being able to sign a message containing the encryption of pms. This acts as a sort of signature from C on pms, but is not enough on its own because signing an encryption does not prove that you know the plaintext. However, C establishes this knowledge by sending a correct "finished" message at the end (where pms should be used as key).

It is an important point that the parties send a MAC on *all* messages sent in the protocol, including in the hand-shake. By MAC'ing the entire view, each party can check that the other party had the same view of the protocol, i.e., the attacker did not change any messages! This essentially forces the attacker to only look at messages sent by the peers and forward these truthfully, which ensures that the overall protocol is secure if the basic protocol —without the MAC— is secure against an adversary that is only allowed to look at the communication. This techniques of *final authentication of views* is very powerful and is used by several key exchange protocols.

Nothing we said so far really proves that the protocol is secure from the type of interleaved protocol attack that killed Needham-Schroeder. It is possible to give such a proof, but this requires a formal model and definition of what security of an authenticated key exchange protocol means. This is outside the scope of this course so we will not give any formal proof here.

But we can argue in a little more detail, as follows: Let us first see things from the point of view of an honest client C who does the protocol with the intention of talking to S. C will send pms encrypted under pk_S, and only S will be able to decrypt this. Furthermore the protocol never asks S to send pms anywhere, not even in encrypted form. Therefore, when C receives a message with a MAC that verifies using pms, he knows this MAC comes from S. If furthermore the MAC can be verified w.r.t. all messages C has sent and received it must be the case that S has seen this same of messages on his side. Therefore, any adversary that sits between C and S can have done nothing except forward all messages unchanged. If the encryption used is secure, this of course cannot lead to a successful attack.

Conversely, let us see things from the point of view of an honest S who wants to talk to C. He sees a signature from C on a ciphertext c and the nonces, and since he chose one of the nonces himself, he knows the signature and hence c was indeed produced now by C, i.e., it is not a replay. Furthermore, an honest C will send a pms to only one server, and S will not himself send pms to anyone. Therefore, when S receives a message with a MAC that verifies using pms, he knows this MAC comes from C. If furthermore the MAC can be verified w.r.t. all messages S has sent and received it must be the case that C has seen this same of messages on his side. Therefore, as above, any adversary that sits between C and S can have done nothing except forward all messages unchanged.

14.3.2 Forward Secrecy and the Future of SSL/TLS

The version of the handshake protocol we have shown here is usually based on RSA for encrypting the pms, but this is not the only version that exists. Another variant is also allowed, that is based on so-called Diffie-Hellman key exchange, which we will explain in more detail in the section on IPSEC later.

There is a potential difference in security between the two variants, which is related to a notion called *perfect forward security*, and this difference was debated intensely after a problem with an implementation of SSL has been discovered, namely the so-called heart-bleed bug. We will have more to say about heart-bleed later, for now we can just note that the bug potentially might allow an attacker to get hold of a server's (long-term) secret key - although no hard evidence is currently known that this actually happened to any web server in real life.

To see the possible consequences of this, consider an adversary who eavesdrops and records an execution of a hand-shake and the following exchange of encrypted data. Now, if the adversary later gets hold of the sever's private key, he can clearly decrypt *pms* and then decrypt all the data. What this shows is that the variant of the SSL hand-shake we have seen is not forward-secure.

Indeed, forward security means that the security of the session you do now cannot be compromised even if the adversary steals a long-term secret key some time in the future. The variant of the hand-shake that is based on Diffie-Hellman is in fact forward secure, so therefore the Internet Engineering Task Force which is responsible for maintaining the TLS standard has stated that they will remove the RSA based version of the key exchange in the next version.

14.4 IPSec

Making a long story as short as possible, IPSec is a set of protocols that do a job very similar to what SSL can do, but on a different layer in the standard protocol stack, namely inside the transport layer. As suggested by the name, this is a set of protocols for setting up a secure connection, also known as a *security association* between two ip addresses. The fact that the protocol is on a lower layer means, for instance, that the sequence numbers and other control information that is used to handle a TCP connection are encrypted while in transit, this would not be the case for SSL. On the other hand, data from different http connections typically go through the same tunnel if they travel between the same IP numbers, so this gives in itself no separation between different applications that run on the same client, for instance.

IPSec is designed and maintained by the Internet Engineering Task Force (IETF), and can handle connections using the new IP v6 addresses (IP v6 is the new standard that aims to solve the long standing problem that there are not enough different IP addresses in the old standard).

To create a security association, IPSec uses a key exchange protocol known as the Internet Key Exchange (IKE). IKE is actually a name for a rather large set of different alternative methods. Typically, however, public-key certificates are used to identify the other party, just like in SSL. Unlike SSL, however, the main option in IKE does not use standard public-key encryption and signatures, based on RSA for instance. Instead, a different public-key technique is used, known as the *Diffie-Hellman key exchange*. Note, however, that IKE is designed to be independent of the rest of IPSec, so any authenticated key exchange protocol can in principle be used without having to change the protocols for data transport in IPSec.

14.4.1 (Authenticated) Diffie-Hellman Key Exchange

Diffie-Hellman key exchange in its basic form uses arithmetic modulo a prime number p. Some number g in the interval from 0 to $p-1$ is chosen once and for all, and when client C and Server S want to exchange a key, we do the following:

1. C chooses a random number a and sends $g^a \bmod p$ as well as his certificate to B.
2. Likewise, S chooses b at random and sends $g^b \bmod p$ as well as his certificate to A.
3. C computes $(g^b \bmod p)^a \bmod p$ based on what he got from S and his own random choice. Likewise, S computes $(g^a \bmod p)^b \bmod p$. These two values turn out to be equal, namely they are both equal to $g^{ab} \bmod p$, so this value can be used as a common key.
4. C signs all messages he has seen so far and sends the signature to S. Likewise S signs all messages he has seen and sends the signature to C.
5. Both parties verify the received signature against the messages they have seen and the public key of the other party. If everything is OK, the protocol was successful.

The fact that that the parties agree on a common key follows from basic rules for exponentiation, namely, if we forget about the divisions by p, it is well known that $(g^a)^b = (g^b)^a = g^{ab}$. It turns out that the reductions modulo p make no difference for this rule, equality still holds. A third party observing the protocol will only see $g^a \bmod p$ and $g^b \bmod p$. If p is chosen large enough, it is believed to be infeasible to compute from this information the key $g^{ab} \bmod p$. This is the so called Diffie-Hellman problem.

It should be noted that, just like for SSL, we give an “idealised” version of the protocol where we abstract away some details that are found in the actual standards.

It is now easy too see why Diffie-Hellman key exchange is forward secure as we stated earlier: the long-term secret keys are here only used for signing messages. So the security of the session key we establish is based on the Diffie-Hellman problem and has nothing to do with the private long term signing key.

Issues with Diffie-Hellman key exchange

Some potential problems with Diffie-Hellman key exchange have been pointed out in the literature. These issue arise when the protocol is based on arithmetic modulo a prime, as we have described it here (it can also be based on Elliptic Curve Cryptography, which is a different story). A particular example is found in some versions of SSL (which, as mentioned, can also use Diffie-Hellman). The problem comes from the fact that there is an algorithm known as Index Calculus that will solve the Diffie-Hellman problem, and hence break the protocol. The special feature is that it will be able to do so very efficiently, if one first does a very long and inefficient preprocessing, *that only depends on the prime p*. What this means is that once the preprocessing is done for a particular prime p, then one can easily break any instance of the protocol that uses that prime p.

Now, some versions of SSL has a number of choices for p hardcoded in the software. The idea probably was to make implementation easier, so that the parties would not have to generate and verify primes on the fly. But the downside is that a powerful adversary, such as a national intelligence agency, could attack these primes using Index calculus and then once they succeed, break into lots of secure channels. The Snowden documents mention an alleged "break-through" by the NSA in attacking encrypted communication. There has been speculation by several scientists that this may refer to a successfully completed precomputation as described here, on a 1024-bit prime. This is not unrealistic given current estimates of what it would take to complete such a computation.

There are several ways to solve this problem: either a fresh new prime for the Diffie-Hellman protocol should be chosen, if not for every instance of the protocol, then at least with regular short intervals. But we can also use a different set-up, namely the Diffie-Hellman protocol can be based on so-called Elliptic Curve cryptography instead of arithmetic modulo a prime. Then the Index Calculus algorithm no longer applies and the problem goes away as far as we know.

14.5 Comparison of SSL and IPsec

Should one use SSL or IPsec? The best answer is probably: it depends, since the solutions have different and complementary properties. The differences follow naturally from the fact that the solutions are placed differently in the standard protocol stack, and are best understood by considering exactly who the communicating parties are in the two cases.

For IPsec, the communicating parties will be units with an IP number, for a standard PC or laptop, this would be the network adapter. This means that the "secure tunnel" goes between such IP nodes and as soon as the network adapter delivers data to the machine it sits in, the data is no longer protected. On the other hand, all applications on the machine can use the protected connection without being aware of the encryption.

For SSL, the communicating parties are applications, typically a browser and an Internet server. This of course means that the applications need to know how to run SSL, but on the other hand data are protected all the way from inside the client until it reaches the server. On a multiuser machine, for instance, this can be used to separate applications owned by different users.

14.5.1 Limitations of SSL and IPSec

One should remember that both SSL and IPSec establish secure tunnels between two entities, after the data leaves the tunnel, it is no longer protected. Therefore, if what your application requires is signatures on documents, e.g., to bind someone to an agreement, then neither SSL not IPSec are useful. The situation is the same is you want to store data in encrypted form, since again data are not protected as soon as they leave the tunnel. We mention below some other standards that can be useful for such purposes.

14.6 Password-authenticated key exchange

As mentioned, one flexibility offered by SSL is that one can allow for "one-way" authentication, i.e., the server identifies itself towards the client, but no authentication is performed of the client. One-way authentication is the best you can do if the client does not have a certificate, as is often the case in practice. It can still accomplish something useful, however: having established a connection, the server still has no idea who it is talking to. However, the client can now authenticate itself by sending a password to the host over the secure connection. Since the server is authenticated and the channel is encrypted, the client knows it will not be revealing the password to a malicious server.

The "one-way SSL plus password" method is useful, but also has some disadvantages. First, it only allows the client to authenticate itself towards the servers it holds passwords for, i.e., passwords that those severs are able to verify from the information they hold. Second, it becomes harder to prove security of the total process because the sending of the password is not an integrated part of the protocol, but comes "after the fact". So some researchers have suggested that in such scenarios the user authentication is only based on security of the passwords anyway, so we we might as well use only the password and forget about the certificates. One should then try to design the protocol such that an attacker cannot break the scheme essentially faster than the time it would take him to guess the password by repeatedly trying to login with guessed passwords. This is simply because such an attack can never be prevented, so the optimal result is to prove that this is the best possible attack.

At first sight, this may seem like a very easy problem: since both parties know a secret (*pw*) can't we just derive a secret key directly from *pw*, or even use *pw* itself for encryption and MAC'ing? However, the problem is that long term usage of such a key on large amounts of data is generally problematic. Even worse, we are talking about a password that must be short enough to be remembered, which means that the number of possible passwords is usually much smaller than the number of possibilities one would want for a secret encryption key. Therefore, if the adversary gets hold of some plaintext encrypted under the password, he can (offline, on his own machine) run through all possible passwords and find the correct one very efficiently, namely the password that leads to a meaningful plaintext after decryption. This is not the security we wanted – as mentioned, we want that the adversary can do no better than to trying to log on-line a few times, which is of course much slower than off-line guessing and can be stopped completely by freezing an account after a few failed log-ons.

So although the problem is not trivial, it can be solved with some care. Password authenticated key exchange protocols are relatively new, but one such protocol, known as SRP, has been standardized by the Internet task force and IEEE.

It has, however, no security proof currently. There are provably secure solutions around, for instance a suggestion by Bellare *et al.* from the EuroCrypt 2000 conference (*Authenticated key exchange secure against dictionary attacks*).

Just to give an idea of how such protocols work, here is a simplified version of the protocol from Bellare et al. (which goes back to another paper by Bellovin et al.). We base ourselves again on Diffie-Hellman, but with a twist, namely we encrypt the messages under the password pw that is assumed to be known by both client and server. The description of the protocol below assumes a successful run where both the client and the server have the same password:

1. C chooses a random number a and sends $A = E_{pw}(g^a \bmod p)$.
2. Likewise, S chooses b at random and sends $B = E_{pw}(g^b \bmod p)$.
3. C decrypts B with his password pw and computes $(g^b \bmod p)^a \bmod p$ based on the result of the decryption and his own random choice. Likewise, S computes $(g^a \bmod p)^b \bmod p$. As mentioned before, these values are equal and can be used as a common key.
4. Here follows some steps where the parties confirm that they agree on the key and have seen the same set of messages. The details are not important here.

Note that if the passwords used by the client and the server in Step 1 are different, then the protocol will not succeed (that is, the client and the server will end up with two completely unrelated keys), since decrypting a ciphertext with the wrong key can be typically assumed to produce just random garbage.

The intuition for the security behind this construction is that an adversary who listens to an execution of the protocol should not be able to brute-force the password just from the transcript of the protocol. Of course the adversary can make some guess at pw and try decrypting both messages. But if we make sure that decryption by any password, correct or not, outputs some number modulo p, then the adversary does not learn anything: any number mod p is as likely to be $g^a \bmod p$ as any other number. So the adversary must spend time interacting with the parties to verify a guess, and this was what we wanted.

This protocol requires both parties to know the password in cleartext. But in most cases the server does not store the password, but instead a hash value or similar computed from the password (as discussed in Chapter 10). Fortunately, SRP and related protocols are able to handle this case as well; they can work even if the server has some encrypted or hashed representation of the password.

14.7 Applications of Secure Channel Set-up

14.7.1 Secure http

Secure http is simply another name for usage of SSL or TLS to set up a secure connection between a web server and a client. Secure http can be referred to directly in web addresses as `https://www....` Such a connection works on a level below the application, that is it creates a secure tunnel between the client and the server, anything that goes through is encrypted and checked for authenticity, and the higher level application does not need to be aware that the connection is secure.

14.7.2 Virtual Private Networks

Virtual Private networks (VPN) solutions are designed to set up secure connections from a local area network to a remote PC. During the set-up, which happens between the PC and a VPN gateway, the remote PC is assigned an IP-address as if it was a part of the local network. After setting up the connection all traffic between the PC and the VPN gateway is encrypted and checked for authenticity. This requires, of course, that an authenticated key-exchange is done during the set-up, using techniques like the ones we saw earlier. Most VPN products use their own proprietary protocols for setting up the connection, or they use the IPSec protocol.

The PC can now work as if it was physically a part of the local network. The idea is that a company can for instance allow employees to work from home, while still having access to confidential company data as if they were at work. VPNs can be also used to prevent location-based censorship.

VPN solutions can certainly be useful, but are not enough by themselves. Even with VPN, one usually needs also a firewall and other protection (see below).

14.8 Higher Level Channels

The protocols above are on a relatively low level: they provide secure communication between machines and low-level processes, not between human users. For example, even an SSL connection where both parties are authenticated cannot be used to sign an email or a document, simply because SSL does not know about the concept of a document, it just encrypts and authenticates everything that is sent.

There are several protocols and solutions around that provide security for higher-level objects. For instance, the S/MIME standard. MIME is the standard format in which emails are transported on the net. S/MIME is an extension where some extra fields are added, that allow transport of encrypted and signed emails. There are reserved fields for a signature, for a secret key that is encrypted under the recipient's public key, etc. Another example of this type is the XML standard – an extension of HTML which, among many other things, allows incorporating signatures and encryption into www documents.

Another way to do "high-level" cryptography is the Pretty Good Privacy (PGP) software (exists also in a public GNU version known as GPG). This is used by many private people, but mostly by those that are actively interested in security. PGP provides capability to sign and encrypt mail and documents. It does not use certification of public keys by CA's, instead it uses a concept known as "web of trust", i.e., your friends can send you a public key of a third person and assert that they trust this public key. You can then decide to trust the key, if enough of your friends have recommended that you do so (or if you have received the key from a source you trust yourself).

14.9 Exercises

Exercise 14.1 Consider the Needham-Schroeder protocol. Suppose we remove the last message from the protocol, i.e., A simply doesn't send $E_{pk_B}(n_B)$. What happens now to the attack described in the notes – does it still work? Does the protocol still satisfy the properties required from an authenticated key exchange protocol? Hint: look at the situation from B's point of view after the protocol is finished.

Exercise 14.2 Authenticated Key-Exchange

Consider the situation where a browser is about to start an SSL connection with a server, as a result of the user having typed a URL to connect to. The description of SSL says that the server should send its certificate and that the client (in this case the browser) should verify it. We assume in the following that the browser will verify the certificate without involving the user, and will only alert him if the certificate doesn't verify correctly.

The certificate often contains the URL at which the server can be contacted, and in this case the browser may check that the URL found in the certificate is indeed the URL that the user typed (in this exercise, to keep things simple, we ignore difficulties that you may encounter in real life, due to the fact that because of redirection, you can sometimes end up at a URL that is different from the one you typed, even if you get the right web site)

Question 1: Argue that this check makes a difference to security: assume the above check is not performed. Now describe an attack where the user wants a secure connection to web site W_a, but instead gets a secure connection the adversary's website W_b without the browser showing any warnings to the user. Your attack is valid even if the browser shows in the address line an address different from what the user wanted - it may well be the case that the user doesn't notice the difference. The threat model includes attacks where an adversary can hijack http connections, i.e. insert/remove messages of his choice, or completely take over a connection.

Question 2: if a URL is not present in the certificate, (or if you do not want to require that the URLs match), the check we looked at can of course not be performed. What could the browser do instead to solve the problem you identified above?

Question 3: The handshake and other protocols in SSL can be proved secure using a number of methods (although the exact definition of security used is not always the same). Yet we have now seen some security problems occurring if the check on the URL (or some other check) is not done, despite the fact that no such verification is required in the SSL specification. This question is designed to give an explanation for this weird phenomenon.

In the beginning of this chapter, we gave a definition of AKE, which we will call D here. Definition D requires that each party starts the protocol with an input that specifies the identity of the other party you want to exchange a key with. One could also consider an alternative definition D' where there is no input, but if the protocol is successful, the output should be a key plus the identity of the party you have managed to exchange a key with.

More precisely, D' is the following definition:

Agreement If A outputs (K_A, identity B) and B outputs (K_B, identity A), then $K_A = K_B$.

Authentication and Secrecy if A outputs (K, identity B), then B participated in the protocol, and he must output either (K, identity A), or "reject". Furthermore, if no party outputs "reject", the adversary does not know K.

Freshness Same as in D (and not relevant for this exercise).

Suppose you are given an AKE protocol π' that satisfies definition D'. Show how to construct from π' a new AKE protocol π that satisfies definition D. You are looking for a very simple solution in which your protocol may fail even if π' succeeds.

Consider the SSL AKE without the URL check. Which of the two definitions D, D' does it satisfy?

Exercise 14.3 Consider the "abstract" SSL handshake protocol from this chapter. Although there might well be simpler solutions to the same problem SSL solves, all the elements in the protocol are there for a reason, and this is the subject of this exercise.

First, suppose the nonces n_S, n_C are not sent, and the session key is derived only from *pms*. Suppose that after a long time, an adversary gets hold of a session key that has been used earlier, for instance by exhaustive search. Since the session key is not in use anymore, this ought to be harmless. However, show that the adversary can now successfully start a session with S, pretending to be C. Explain why this would not be possible with the original handshake.

Second, suppose we remove from the protocol the MAC's computed over the previous communication (which are normally sent in the steps 5-6). Show the following: when honest client C makes a call to S, an adversary can intercept the call, and make C believe that the protocol completed successfully, when in fact S was never involved.

Exercise 14.4 An early version of Netscape implemented the client side of the SSL protocol as follows: it took the local system time, which could be obtained to a precision of 1 microsecond. Then this was used as input to a fixed function (known as a pseudorandom generator) and the output was used as the pre master secret *pms* in the SSL handshake.

What do you think of the security of this idea? Give an estimate of the time needed to break it, as a function of the time Client and Server would normally spend to do the handshake once.

Exercise 14.5 Consider the following proposal for set-up of a secure channel: Assume A has already the public encryption key pk_B of B. Assume also that ciphertexts made with pk_B are from the same set as the plaintexts – as is the case for RSA, for instance.

1. A chooses a random message R and sends to B $E_{pk_B}(E_{pk_B}(R))$.
2. B decrypts twice to get R and sends $E_{pk_B}(R)$ back to A
3. A checks that the value he received is indeed $E_{pk_B}(R)$, and if so he sends OK to B.
4. Both parties now use R as a secret key for the following communication. Say, we derive from R both a key for encryption and for message authentication codes, and every message sent is encrypted and tagged with an authentication code.

Suppose first that B is honest, and receives OK in step 3. When B receives a message after step 3 with a correct MAC, can he trust that this message came from A? When B sends an encrypted message, can he trust that only A will be able to decrypt? Why or why not?

Suppose now that A is honest and the check in step 3 is OK. When A receives a message after step 3 with a correct MAC, can he trust that this message came from B? When A sends an encrypted message, can he trust that only B will be able to decrypt? Why or why not? Hint: the answer to this one is more subtle that you may think. An adversary could start a new session with B and use data he knows from the past.

Exercise 14.6 This exercise is about a problem in the first version of the WEP standard, i.e., the protocol that is used to establish secure channels on wireless networks. WEP assumes that sender and receiver have a preshared key K, so no public-key technology is used. Instead, the protocol uses the stream cipher $RC4$, which is basically a function that takes K as input and outputs a random looking string that can be as long as one wishes. This output is used as a one-time pad on the message, so that encryption of message M will be $M \oplus RC4(K)$. The protocol has some built-in procedures for changing the key so that the same output string will not be used on different messages, but this is not relevant here.

The designers of WEP also wanted to ensure authenticity. For this, they used a so-called cyclic redundancy check (CRC). This is a function that takes any string as input and outputs a fixed length string $CRC(M)$. It is normally used to check for transmission errors coming from naturally occurring noise. One sends the concatenation $M||CRC(M)$, and the idea is that if M or $CRC(M)$ is modified, there is a good chance that the last part will no longer be CRC applied to the first part, so that errors can be detected.

Question 1: Explain why this way to use CRC (without encryption) will not help to obtain authenticity.

The idea of WEP was to send the encryption $RC4(K) \oplus (M||CRC(M))$, and this was then supposed to ensure both confidentially and authenticity. Probably, the designers of WEP believed that the problem with authenticity addressed in the previous question would not occur here because of the encryption. They were wrong because they did not take into account the way the CRC function was defined: each bit in $CRC(M)$ is in fact the x-or of a fixed set of bit positions in M, for instance, the first bit in $CRC(M)$ might be the xor of bits number 1,3,5,7, etc of M.

Question 2: Explain how an adversary can modify an encrypted message so that the underlying message is changed, but the CRC-check after decryption will still be OK.

Exercise 14.7 A story from real life on setting up a secure channel: Suppose we want to design a system that can detect automatically when a goal is scored in a soccer game. There is equipment installed in the ball and in the goals, that we assume will detect reliably when the ball goes inside the goal. The problem in this exercise is to secure the communication between the sensor in the goal and a device that the referee is carrying around his wrist. The idea is of course that the referee should be able to tell immediately if the ball really passed the goal line. The security policy makes the following requirements:

- Even if outsiders eavesdrop the communication (which is easy since it is wireless), it should not be possible to use the information learned to fool the referee into thinking that a goal was scored when this is not the case. Likewise, it should not be possible to remove a signal that a goal was scored without this being detectable.
- Even by monitoring continuously the data sent by the sensor in the goal, an outsider should not be able to tell whether the sensor is signaling a goal or not. This is to prevent outsiders from making complaints that the referee ignored a signal.
- To have a practical system, the device carried by the referee must be small and limited in computing power, so the solution must be as simple as possible.

Design a way to handle the communication that will satisfy the above security policy.

You may assume that both the sensor and the referee's device can be initialized with fresh data by an official before the game starts. Is it possible to have an unconditionally secure solution? - that is, one that remains secure, even if the adversary can search exhaustively for any fixed secret key.

Exercise 14.8 Consider the following authentication protocol for two parties S, R who are assumed to share a secret key K_{SR} for symmetric encryption. The protocol is not designed to do exchange of a secret session key, but only tries to ensure for S (R) that R (S) is currently active at the other end of the communication channel.

1. S sends a nonce n_S to R.
2. R chooses a nonce n_R and sends $E_{K_{SR}}(n_R), E_{K_{SR}}(n_S)$ to S.

3. S decrypts under K_{SR} what he receives and checks that the last encryption of the two contains the n_S he chose in Step 1. He returns the result of the first decryption to R.
4. R checks that what S returned equals the n_R he chose in Step 2.

The intuition is that R would not be able to encrypt n_S correctly, nor would S be able to decrypt n_R correctly, without knowing K_{SR}. And so (hopefully) the protocol would not be successful unless both parties were active at the same time. However, consider the following application of the protocol:

A is a machine which communicates with several parties, so it stores for each other party B a secret key K_{AB}, and we assume that B is the only one, other than A, who knows K_{AB}. When another party B wants to start a session with A, the following happens:

- B sends his username to A.
- This causes A to start a new process which will handle the new session. The process looks up K_{AB} in A's database, and conducts the above protocol with B, where A sends the first message, i.e., A plays the role of S and B plays the role of R.

When A wants to start a session with another party B, the following happens:

- A starts a new process which will handle the new session. The process looks up K_{AB} in A's database, and sends the user name of A to B.
- The process then conducts the above protocol with B, where B sends the first message, i.e., A plays the role of R and B plays the role of S.

Question 1: Show that if the processes run by machine A are independent, i.e., do not communicate with each other, then when A makes a call to B, an adversary can intercept the call, do further interaction with A, and complete the protocol successfully, i.e., A will think he is talking to B, even though B is not present at all.

Question 2: How can the way A handles the protocol be changed such that the security problem you identified above is avoided? if you see several ways to solve the problem, feel free to mention them all.

Exercise 14.9 Implement the Authenticated Key Exchange from IPSec as described in the lecture notes. Recall that you have already code from a previous exercise that can do digital signatures.

You do not have to implement certificates, you can have the parties simply send their public keys instead of the certificates (although this of course would not be secure in real life). The fixed parameters (the prime and the fixed base) you can choose once and for all and hardwire them into your code.

Your solution should be able to run using real communication between two machines, recall that you have code from the distributed systems course that can do such communication.

You should verify that the protocol indeed generates the same key for both parties.

15

System Security Mechanisms (DRAFT)

Contents

15.1 The Trusted Computing Base

Whereas network security is about securing communication from one system to the other, system security is about defensive mechanisms that try to prevent external attackers from getting into the system, and also about mechanisms for preventing attacks from parties who are already users of the system.

So the typical scenario here is that some party wants access to some part of our system, and we want to ensure that access is only given in cases where this is OK w.r.t our security policy. Here access should be understood in a broad sense: getting access to a machine does not necessarily mean physical access, but could also just mean that I am allowed to communicate with it.

Clearly, one needs to be able to identify the party who wants access, and we already discussed several issues that are relevant here w.r.t. identification. For instance, password security, hardware tokens and biometrics, but also authenticated key exchange is a way to ensure that we know who we are talking to.

However, identification is not enough. A secure system also needs to have some part that takes the decision on whether some action should be allowed according to the security policy, and also enforces the decision: if access is denied it should not be possible to circumvent the system and get access anyway.

It is obvious that we need to assume that this part of the system cannot be modified by attackers or malicious users. This part of the system is therefore called the *trusted computing base* (TCB), i.e., it is some part or parts of the system that is trusted to perform according to specification, even under attack.

A TCB can be established, for instance, using hardware security and/or software security, depending on the threat model you are in. Concretely, the TBC will often be part of the operating system's kernel, but it may also consist of one or more hardware units. So one should not expect that the TCB is physically located in one place.

Analysis of a system very often starts by trying to identify those part of the system that act as TCB, and estimating whether they can be assumed reliable enough under the assumed threat model.

In the following, we look first at how to protect a system from outside attacks and then turn to security policies for controlling the system's behavior towards inside users. All this can be thought of as descriptions of how different parts of the TCB can be designed.

15.1.1 An example: SGX

A recent example of a hardware based TCB is Intel's SGX. This is a feature that can be found on many newer Intel CPUs, and it is basically a hardware protected area on chip that is capable of doing a full range of CPU instructions and can also permanently store a small amount of data. This is known as a *secure enclave.* An enclave is designed to make it infeasible to read internal data from the outside nor can you make the enclave behave differently from what it was designed to do. Thus an enclave is protected even against the owner of the machine it sits in. An enclave can store internal data outside in insecure memory but will then encrypt and authenticate the data it sends out.

There are many possible applications: for instance, an enclave can be set up to only execute software that has been signed by certain parties. It also holds private signature and encryption keys so it can execute a program P and produce a signed version of the output. This means other parties would get evidence that the data they see came from a correct execution of P, even if they do not trust the owner of the involved machine. The same type idea can also be used for what is known as *remote attestation*: a software vendor could send code to a machine it does not fully trust, perhaps with essentials parts of it encrypted. The code would then run inside an enclave which would be able to check that the software is genuine (against a signature from the vendor) and could also provide an anonymized proof that a genuine enclave has executed the program. One can think of this a a signature that only proves that some enclave authenticated the data, it does not show which enclave did it. This is to protect the privacy of the user.

All this holds great promise in many ways, but one has to remember that the idea also has certain inherent weaknesses: First of all, the integrity of the whole system hinges on the integrity of Intel's organization. For instance, they are the only ones who can install the initial signature key in an enclave and distribute the corresponding public key. Second, when an enclave uses external memory which is often necessary due to the small size of internal memory, the access pattern can be observed, even if the data itself is encrypted. Being able to see which parts of its memory a program accesses and when can often reveal a lot of information.

Finally, in 2018 a serious attack against SGX was published, the so called Foreshadow attack. It is closely related to the Spectre and Meltdown attacks that we will take a close look at in a following chapter. It shows that current implementations of SGX are not secure and it is possible to read secret keys from an enclave - which of course completely breaks the idea of remote attestation. This does not mean that the idea of SGX is fundamentally flawed, but is s symptom of the fact that Intel seems to have put too much priority on speed over security in the current design.

15.2 Firewalls

We have seen many ways to set up secure channels over insecure networks, but this actually does not necessarily help your *system security* very much:

Of course, by setting up a secure channel between A and B, we can make sure that no third party can break into the communication. But even if A will only set up secure communication with parties it knows and trusts, the Internet is an open place where anyone can at least try to communicate with anyone else. At the start of the communication set-up, A of course does not know who he is talking to, in fact the purpose of an authenticated key exchange is to make sure who the other guy is. Conclusion: even a computer who insists on doing authenticated key exchange with anyone who tries to connect will have to physically do some amount of communication with potentially untrusted parties.

This opens the possibility that an adversary can hack into A's computer, using for instance an overflow attack (which we discuss in detail later in the book). With current state of the art, it is very difficult to make sure that such hacks are *never* possible. This sets the stage for the simple idea behind the *firewall* concept: the adversary cannot break into a computer he cannot communicate with.

15.2.1 Packet Filtering

So from a high level point of view, a firewall is simply a machine or a piece of software that stops and removes part of the traffic on a network. It is typically placed on the connection between a company's local network and the Internet, so that network traffic must pass through the firewall to go from the Internet to the local net, or vice versa. The simplest use of a firewall is to make it filter out all communication trying to connect to machines for which you do not want any communication with the outside world.

To understand how this is possible, recall that the connection oriented TCP protocol requires that particular types of packet must be sent and acknowledged in order to open a connection, and such packets can easily be recognized because a certain flag in the packet is set. Later packages can be recognized as belonging to a connection established earlier. Another protocol that is used in some cases instead of TCP is the UDP protocol. This is not connection oriented, one basically just sends a packet and hopes for the best. Among other things, UDP is used for communicating with DNS servers, which are servers that know how to translate human-readable names of Internet servers to actual IP addresses. Both UDP and TCP packets can always be recognised.

The simplest tool we can use in a firewall is called *packet filtering*. The definition of packet filtering is as follows: the firewall looks at each packet and decides whether the packet should allowed through, based only on the content of the packet and some fixed set of rules.

Thus the packet-filter firewall can, for instance, just remove all packets trying to open a TCP connection to a local machine and stop all UDP traffic.

Although this may seem like a simple way to get rid of most risks, it is usually a bad idea, however. Typically, a company has machines for which you definitely want communication with anyone else, for instance the machine running the web-server. Clearly the firewall should not stop such communication, so we would have to make the firewall allow for incoming connections to the webserver. Moreover, "inside" machines that want to use an external web service for perfectly legal reasons, and therefore need to communicate with DNS servers using UDP, should be able to communicate. This can be done by packet filtering, because we can determine which incoming connections is for the web-server, and which UDP packets are meant for or are coming from a known DNS server.

However, this also opens us to attacks: if the attacker can break into the web-server, he can move from there to other machines. More precisely, attacking other machines now only requires sending messages from the web server on the local net, and such local traffic is not stopped by the firewall. More advanced solutions use two firewalls, where the web-server is behind a not so strict firewall, while the rest of the system is protected by a more strict filter. Still, this does not completely remove the problem.

Summarizing on packet filters: if simple packet filtering is to be secure, it needs to be so restrictive that it is hard to live with. If we make it less restrictive, we open a possibility that an adversary can try to communicate with and attack machines on the local net. This is typically done by scanning through a large interval of IP addresses and port numbers until one finds a machine that is willing to respond. This is also sometimes known as "port scanning".

The conclusion is that a firewall needs to do something more intelligent than just a simple packet filter, it needs to examine packets in more detail and either take action that depends on the purpose of a packet, or even modify the packets.

15.2.2 Proxy Firewalls

One radical approach to "intelligent filtering" is a *proxy firewall.* Such a firewall will not allow any connection at all from a local machine to the outside or vice versa. Instead, it will handle any connection needed *on behalf of* the local machines. That is, if a local machine has some client software that wants to use a service on the net, the client must be configured to connect to the firewall instead, and then the firewall will connect to the service. This means that from the outside, the local network is completely hidden, it seems like only the firewall is there. Thus scanning through a large number of IP addresses until some machine out there reacts will not work here. You will never get response except from the firewall, so you are forced to hack the firewall before attempting anything else. Making sure that a particular machine cannot be hacked is hard enough, but at least easier that having to worry about dozens of local machines.

Therefore, proxy firewalls are regarded highly secure. But it is of course not a very flexible solution. For instance, suppose we want to use client software for some web service on a local machine, but this software doesn't know how to use a proxy. The result will be that the client insists on connecting to the site where the service is placed, but this connection will be refused by the proxy firewall, who on its side insists that all local machines must connect to the firewall and no one else. Of course, we may be able to make the client believe that the service can be obtained from our firewall. Then the client will connect as it should, but now the firewall (or some front end that we have to build) has to know to where this particular type of connection should be routed. In other words, we will be forced to do some special implementation or configuration for every type of connection we want to allow.

15.2.3 Stateful Firewalls

It can be a better solution to use a *stateful firewall.* Such a device keeps track of every connection going through it. Whenever someone tries to open a connection it checks whether this type of connection is allowed and blocks it if not. All other packets are checked to see if they belong to an existing connection and are blocked if not (where, however, some flexibility is of course needed for UDP packets). This of course requires the firewall to remember which connections are active at the moment, and this is why such firewalls are called statefull.

With such a firewall, it is even possible to translate the addresses used by local machines to something different in the outside traffic, thus hiding the local addresses from an outside attacker, this is sometimes known as a *masquerading firewall*, and provides another way to avoid the "IP address scanning" mentioned earlier, or at least make it harder. A final step further is to have the firewall inspect the behavior of a connection to see if it does something suspicious, and either stop it or drop packets that have suspicious content. For instance, in some attacks on web servers, the attacker would need to send an unusually long input string to a particular routine running on the server. The firewall can be configured to catch such traffic and remove it.

The final word on firewalls is a word of warning: one advantage of firewalls is that they provide a way to do filtering of network traffic by just administrating one machine, instead of having to do so on every single machine in the local network. Of course it would be wonderful if firewalls provided an out-of the-box and maintenance-free solution to this type of problem. While this may be the case to some extent for personal firewalls protecting only one PC, it is *never* the case when protecting larger corporate networks. The optimal configuration of the firewall of course depends on the configuration of the local network, and hence it has to be updated as the structure of the local network changes. In other words the statement "we installed a firewall, therefore we are secure" is wishful thinking if no one is maintaining the firewall.

15.3 Malicious Software and Virus Scanners

One of the things attackers would want to do, if they make it past the firewall and other defenses, is to install malicious software on the attacked machines. Such software comes in many forms:

- Trojan horses: programs that appear to be useful (or go completely unnoticed by the user), but have some hidden malicious purpose, such as granting unauthorized access, reveal private data, or destroy data. A well-known special case is Spyware: programs that monitor peoples' Internet usage and send information on this to the creator of the program – who can use this information himself, or sell it to a third party.
- Viruses: infect programs on your computer, spread themselves from one program to another, and eventually does something harmful to the machine.
- Worms: similar to viruses, but exist as stand-alone programs that actively try to spread themselves from one machine to the other.
- A final type that does not completely fit into the above categories is *ransomware*: this is a program that initially acts as a trojan, but at some point, it will encrypt all your files under a public key for which the attacker holds the secret key. It will then tell the user that the files will only be unlocked against payment, usually in bit-coin or similar, so the attacker can get the payment anonymously (although bit-coin is not really anonymous).

If malicious software makes it into your computer, you need a tool that can recognize such programs and remove them. This is what the well known virus scanners do, they basically scan all potentially dangerous files and locations on your harddisk, looking for pieces of code that is known to be suspicious. This is possible given that a database has already been gathered, containing information on a large number of known viruses, typically in the form of *virus signatures*, particular bit patterns characteristic for particular viruses. Virus scanners are only useful if their databases are kept up to date. The largest producers of virus scanners are typically remarkably fast in updating their data files when a new virus hits the net. Unfortunately, many users do not update their own copy of the scanner, or do so too slowly. And hence many infections happen because of outdated virus scanner files and not because of new viruses. In recent years, various forms of malicious software have started trying to fool virus scanners by encrypting themselves in order to not look like what is in the scanner's databases. This can work if the virus contains a small piece of cleartext code that will decrypt the real virus and load it in memory just before execution.

An additional precaution that is very simple but also crucial is back-up: many of the attacks, ransomware in particular, lead to loss of data, and the back-up may be your only hope of getting your data back. Note that convenient solutions where your machine has access all the time to a back-up disk and backs up automatically will not help against randomware: the attack can encrypt all files it has access to, including the back-up!

How does malicious code make it into your machine in the first place? Even if an attacker can talk directly to a machine, he can usually not immediately install any software he wants, in particular if he is not a registered user of the targeted machine. There are several ways around this problem, however: the attacker can try to exploit weaknesses in the operating system or server software via, e.g., overflow attacks or similar. We look in detail at this later in the course. This can lead to attacks that will work if a user just visits a particular web page, provided his operating system has some known and exploitable bug. It is important to understand here that a web page is not a passive object you just look at, a web page may contain code that is executed as soon as the page is shown, for instance JavaScript and the like, leading to potential problems if the code has been maliciously designed.

In many cases, however, it is just as easy to fool the user into installing the malicious code himself! This is a special case of social engineering that we discussed earlier. For instance, there are millions of web pages with pop-up windows that claim to have identified a problem on the user's machine and offer to fix problem for free, if the user clicks OK to installing some piece of software. Surprisingly many users fall for this, and it is important to understand that firewalls and the like cannot help much here, the only hope is that the anti-virus or anti-spyware software will recognize what you are about to install.

15.4 Intrusion Detection

A different way to spot malicious software or users is to try to detect malicious behavior, rather than the malicious code itself. That is, we monitor processes and/or users of a system and try to detect if their behavior is suspicious. The big potential advantage is that this method does not have to depend on information on already known viruses and/or attacks.

We already saw one example of this: a stateful firewall can keep track of the connections going through it, and can therefore detect if the client sends data of suspicious form or size to a server. For instance, if a server is expecting input data coming from web forms that users are supposed to fill in, then data strings are usually supposed to be of at most a certain length. If an input string is much longer, this may be caused by an attack where a malformed input string is being sent in the hope that this will make the server behave in some unexpected way that can be exploited by the attacker.

Other form of this look at specific aspects of the behavior of users or programs, such as login and session activity, command or program execution activity, or file access activity.

There are two approaches to deciding whether something suspicious is going on, known as *rule-based* and *statistical* intrusion detection. In the first case, we set up in advance a set of rules describing what normal behavior is, and anything that is significantly different is considered illegal. In the second case, the idea is to first gather statistics on actual honest behavior and then compare to the statistics of the observed user or program.

The basic problem with intrusion detection is similar to basic problem with biometrics: the entity monitoring the system must be tolerant enough to allow legal behavior, but restrictive enough to catch suspicious behavior. At the time of writing, this problem cannot be said to be completely understood or solved. Another issue is that the entire idea is no good, if the system does not react when an attack is detected. Thus, the part of the system that alerts the system manager, closes the system down, or whatever, is as critical as the part that detects the attack.

A final very useful trick in the intrusion detection area is the so called "Honey-pot" approach. The idea is to create a resource on your system that will look interesting to an attacker but is actually fake, that is, a resource that no honest user would want to use. As a result, if activity involving the honey-pot is detected, we can assume this is adversarial activity. By logging such traffic one can hope, for instance, to trace it back to the attacker before he becomes aware that he is being traced. One known example of this took place at a hospital where it was suspected that someone was illegally accessing confidential files on patients. Following the honey-pot approach, a number of files for non-existing patients were created, and accesses to these files were used to trace the attacker.

15.5 Security Policies

As mentioned in the first chapter of this book, we usually organize the way we think about security into the concepts of

Security Policy A specification of the system in question, a description of the security objectives for the system, and perhaps a high-level strategy for achieving the objectives.

Threat Model A description of the attacks we want to be protected against.

Security Mechanisms The technical solutions we use to reach our objectives.

Whereas the previous chapters have been about Security Mechanisms, and to some extent about Threat Models, we now turn to Security Policies. Here is a more detailed but still informal definition:

A security policy contains a description of the system that the policy applies to. It then describes the security objectives you want to achieve. This can take different forms: it can be a specification of what different entities are allowed to do and not allowed to do. It can also be a specification of which events we want to tolerate when the system runs. The system must then ensure, either that no entity can do what is not allowed, or that no event occurs that we do not want to tolerate. Finally, the policy may contain a high level (non technical) description of the methods that will be used to enforce the security objectives.

Note that it is of course important to say which system we want to protect, i.e., it makes no sense to define what security means for a system unless we also describe the system itself, at least to some level of detail. This also has the purpose of limiting the problem: we do not have to protect something that we have explicitly defined as being outside of our system.

Security Policies are closely connected to some of the elements of system security we described in the previous chapter, most importantly the trusted computing base (TCB) , the "incorruptible" part of a system that is required to protect against insider attacks. The security policy can be thought of as the place where we formulate the rules that the TCB should enforce.

15.5.1 Models for Security Policies

In this section, we look at models for security policies, i.e., general "recipes" for putting a security policy together. This means that we distinguish between

Models for Security Policies An abstract description of ways to design a security policy. Such a model typically does not describe a particular system, but it often addresses particular threat models.

Security Policy Is typically a realization of some model, w.r.t. a particular concrete system. A security policy can also be based on more than one model, or it can even be "unique", i.e., not based on any model.

Before looking at examples, we give a definition of a useful concept which is used in many models, namely a *lattice*. A lattice consists of a finite set S plus a relation$\leq$. For any elements $a, b, c \in S$ it must hold that

- $a \leq a$
- $a \leq b$ and $b \leq a$ implies $a = b$
- $a \leq b$ and $b \leq c$ implies $a \leq c$

As a typical example, one may think of S as the set of different privileges or rights a user may have in system, and $a \leq b$ means that with rights b, you can do at least as much as with rights a. For this interpretation to make sense, we clearly need that the three conditions above are satisfied. However, it may be that some rights are not comparable. Suppose a user may have read- or write-access to a file, or none of these, or both. While it is natural to say that $\textit{nothing} \leq \textit{write}$ and $\textit{read} \leq \textit{both}$, there is no natural way to compare *read* and *write*. Therefore the definition of a lattice does not require that the relation $\leq$ is defined for all pairs a, b.

In order for $S, \leq$ to be a lattice, it is further required that for any $a, b \in S$, we have

- There exists a *greatest lower bound* for a, b, i.e., some $c \in S$ such that $c \leq a$, $c \leq b$ and for any other c' with this property, we have $c' \leq c$.
- There exists a *least upper bound* for a, b, i.e., some $d \in S$ such that $a \leq d$, $b \leq d$ and for any other d' with this property, we have $d \leq d'$.

Note that a and b do not have to be different, and clearly the greatest lower bound of (a, a) is just a itself. Also the greatest lower bound for (a, b) can be equal to either a or b. In fact, if $a \leq b$, then the greatest lower bound for (a, b) is a.

There are (at least) two ways in which one can use the lattice model to organize a security policy:

1. Define *subjects* to be all users and processes in a system, that is, any entity that may actively do something to other parts of the system. On the other hand *objects* are resources, files, hardware devices, etc.. We may now classify subjects and objects such that each of them are assigned some position in a lattice. We can then say that subject s with classification $C(s)$ may do a certain operation on object o with classification $C(o)$ if and only if $C(s) \geq C(o)$ – or if and only if $C(s) \leq C(o)$.
2. We can assign a position in a lattice to different parts of a system and say, for instance, that information may only flow from position a to position b if $a \leq b$. This would make sense, if higher positions mean "more secret" and our policy focuses on confidentiality. If instead we focus on authenticity and higher positions mean "more reliable parts of the system", we could specify that information can only flow from a to b if $a \geq b$. This way of thinking is not completely disjoint from the first item above: if we specify the rules for access in certain way, based on positions in the lattice, this may result in a certain information flow, as we shall see below.

The first item above gives a motivation for requiring in the definition of a lattice that least upper bounds and greatest lower bounds exist. Suppose for instance we have two files, classified as a, respectively b, and the rule is that read access is only granted to subjects with classification at least as high as the object. Then if we ask "who can read both these files?", the answer can be stated in a simple way, namely "all users with classification least upper bound of a, b or higher". If we had not required existence of least upper bounds, the answer to the question could in principle have contained an arbitrarily long list of classifications.

Note that the existence of least upper bounds and greatest lower bounds generalize to any subset of positions in a lattice (see the exercises).

One early example of a model that can be seen as a simple special case of the Lattice model is the Bell-Lapadula model. This was the first ever attempt to formalize a security model. It was conceived for military applications and focuses therefore on confidentiality. The idea is to define a number of security levels, which are linearly ordered, say $public \leq secret \leq top - secret$. The model then defines two rules (sometimes known also as the simple security property and the $*$-property):

No read up: Subject s may read from object o only if $C(s) \geq C(o)$.
No write down: Subject s may write to object o only if $C(s) \leq C(o)$.

The idea with these rules is of course to prevent information from flowing form higher levels to lower ones, and so information on the top-secret level, for instance, is kept confidential from the point of view of subjects on lower levels. Taken literally, the Bell-Lapadula model is very rigid and hard to handle in practice: the system may not be able to do what it is supposed to unless *some* information flows downwards. Also, how does one handle changes of classification? Nevertheless, Bell-Lapadula has served as point of departure for system designs where confidentiality was the most important issue.

The Biba model is similar to Bell-Lapadula, but focuses on integrity. Here, the higher levels are the ones that contain the most reliable information. In order to maintain this, the Biba model has two rules:

No read down: Subject s may read from object o only if $C(s) \leq C(o)$.
No write up: Subject s may write to object o only if $C(s) \geq C(o)$.

The idea here is that information must not flow from lower levels to higher ones. This ensures that highly reliable information is not “contaminated” with less reliable stuff. As an example, think of integrating a traffic control system for trains with a system for informing passengers about delays etc. Clearly, the traffic control system must contain reliable information about the position of trains, so it is natural to require that information can flow from the traffic control system to the information system but not in the other direction.

Both Bell-Lapadula and Biba can be extended to so called *compartmented models*, where the different levels are split in different compartments, and where we do not allow arbitrary information flow between compartments, even if they are on the same level. This leads to more general lattice structures than the simple linear one.

We now briefly mention a few examples of other models that are not directly based on lattice structures:

Chinese Wall This is a model inspired by commercial scenarios, where for instance a company C has several different clients. Some of these clients are competing with each other and we want to design a policy that will ensure that no employee of C can reveal information to a client about a competitor. To this end, we assume a predefined predicate *compete*, where $compete(c_1, c_2)$ for clients c_1, c_2 is true if and only if c_1, c_2 are competitors. The rule is then that subject s is granted access to information about c if and only if he never accessed information on any c' for which $compete(c, c')$ is true.

Note that this rule is dynamic: as soon as s has accessed information on c, this prevents him in the future from accessing information from any competitor of c.

Prevent-Detect-Recover The idea is to split the policy in 3 parts: one part describing how we will try to prevent attacks from happening at all, one part describing how we can detect a successful attack, and finally a part describing we will do to recover once we detected a breach of security.

A typical example of such a policy is virus protection: one can attempt to prevent virus attacks by only accepting mails from certain senders. As this may not be sufficient, one can use virus scanners to detect vira in mails, and finally one can have a part of the policy saying that an infected mail is automatically deleted.

Separation of Duty Is a model that comes in two flavors:

Dual Control is a model where certain actions are defined to be possible only if they have been authorized by several persons or entities. A well known example is the use of nuclear weapons, as seen in countless movies.

Functional Separation is also a model where an action requires involvement of several persons or systems, but at different points in time. An example is signing up for an exam, where the instructor has responsibility for evaluating the mandatory exercises, and report to the lecturer, who in turn reports to the study office. In addition, a student must also sign up himself.

15.6 Access Control

The models we have seen so far allow us to describe on a high level what flow of information or control between different persons or parts of systems we want to allow. It is implicitly assumed that when a user wants to execute an operation on some object, the TCB is able to verify the identity of the user and figure out which rights and privileges he has. Checking the identity of someone is a technical problem that has to do with password and hardware security or even biometrics, and has been treated earlier. Figuring out which rights a given user has at a given time is a different problem. In principle, this can be computed from the user's identity and the security policy. However, doing so on the fly —while a user is waiting for access, for instance— requires that the information about this is organized in a appropriate way. So the way we treat access control here can be thought of as a treatment of how we can represent the rules implied by the security policy in a convenient way so it can be accessed and used by the TCB.

The most generic way to represent the information is known as the *Access control matrix.* This is a matrix A with a row for each subject s and a column for each object o. Entry $A[s, o]$ contains a list of all operations s is allowed to execute on o. On most realistic systems, storing the entire matrix in a central location is not practical. So in practice, two basically different approaches are taken:

Access Control Lists Here we store, together with each object, the column assigned to the object, and this is called an access control list (ACL). Such lists make it easy to decide for a given object who has access, but harder to say what a given subject can do. For instance, this is basically the approach of the UNIX system.

User Capabilities Here we store, for each subject, the row assigned to the subject, this is typically called a user capability. They make it easy to compute what a given user can do, but harder to decide who can access a given object. For instance, Windows uses this basic approach.

For large systems, ACL's get much too long if they are specified per subject (user), and so one typically splits subjects in predefined groups and the ACL says only what subjects from each group can do. For instance, UNIX splits subjects from the point of view of a single user, into: the user himself, the user group he is in, and anyone else. Groups also make it possible for several subjects to share the same list of user capabilities.

A variant of the user group idea is known as *role-based access control*. Here, a number of roles are defined, such as "ordinary user", "administrator", etc. and capabilities are specified for each role. Users are assigned a certain role when they log in and are granted access according to the role. This is slightly different from the user group approach in that the same physical user may play different roles at different times.

15.6.1 When the Access Control Matrix is Dynamic

A final issue on access control is how the access control matrix is updated. Who can do this, and when? Basically, we distinguish two approaches: *mandatory access control*, where subjects cannot change the matrix, and *discretionary access control* where subjects have the power to update the matrix, or at least certain parts of it.

Given that the access control matrix may change over time, it is natural to ask what types of information flow this may allow. For instance, a natural question is: can a certain access right r ever be added to some cell $A[s, o]$ that did not contain r before?

Harrison, Ruzzo and Ullman have studied this question in a formal model, where certain *primitive operations* on access control matrix A are allowed, namely: create/delete subject s, create/delete object o and add/delete access right r from $A[s, o]$. It is assumed that there is a fixed finite number of different access rights. A *command* has the following format:

```
Command com(parameter-list)
  r1 is in A[s1,o1]
  ..
  rn is in A[sn,on]
  list of primitive operations
```

The meaning is that the list of primitive operations will be executed if all the conditions in the first part are satisfied. All the rights, subjects and objects referred to in the specification of *com* are taken from the parameter list that is filled in with concrete values whenever *com* is executed.

Now, suppose we are given a fixed set of commands $com_1, ..., com_t$ and an initial access control matrix A. Now we can ask, the above question in a formal way: for given access right r, does there exist a set of commands that will transform A into a matrix A', where $r \in A'[s, o]$ for some s, o but where $r \notin A[s, o]$?. If the answer is no, we say A is *safe* with respect to r.

One may now prove:

- If commands may contain more than one operation, then it is undecidable whether A is safe w.r.t. r.
- If all commands only contain a single primitive operation, then it is decidable whether A is safe w.r.t. r.
- If there is a upper bound fixed ahead of time on the number of subjects and objects, then it is decidable if A is safe w.r.t. r.

Some explanation of what these result mean: a decision problem (the answer is yes or no) is *undecidable* if it is the case that any program trying to solve the problem will either run for ever or will in some cases give an incorrect answer. A famous example of such a problem is the *Halting problem*: you are given a program and your task is to decide if the program will always halt, no matter which input it gets.

In realistic systems, one should expect that commands will contain more than one operation, and it does not always seem realistic to assume that there is a bound on the number of subjects/objects defined ahead of time. So the results sounds rather pessimistic, since an undecidable problem is intractable in a very strong sense. One should remember, however, that the undecidability only asserts that there is no general procedure that will answer the question *for any system*. In a given concrete case, we may still be able to get an answer. However, the result does illustrate in a striking way the formidable complexity you may run into when analyzing a dynamic access control system.

The proof of the undecidability result works by noticing that what we have is a state machine, where the initial state is A and the current state at any given time is the current value of the access control matrix. The state can change by executing a command with some given set of parameters. It now turns out that by designing the set of access rights and commands in a clever way, we can make the system emulate the execution of any given Turing machine T on some input x – in such a way that the question of whether A is safe w.r.t. s is exactly the question of whether T halts on input x. The result then follows from undecidability of the halting problem.

Therefore the conclusion is that when designing dynamic access control systems, the best advice is: keep it simple! – there is good theoretical evidence that even moderately large systems of this type can be very hard to analyze.

15.6.2 An example: Java

As an example of security policies, one may look at how security is handled for applets written in the programming language Java.As is well known, applets are (small) programs that users typically get by downloading from the web, which clearly leads to security concerns since an applet may be malicious.

In the first version of Java, all applets had to run inside a *sandbox* which is a memory area that is separated from the rest of the system. Access outside the sandbox was/is prevented by the virtual machine (VM) which itself a program that interprets and runs the applets. Thus in this set-up the VM acts as the TCB. In the next version (1.2) it was realized that this in many cases was preventing applets from being useful because nothing could be done that involved the user's own files. And so signed applets were introduced. Now, a signed applet was allowed free access outside the sandbox.

In the final version treated here (1.3), it was realised that this was too inflexible, and fine-grained security policies were introduced. Here the user can write a security policy file that specifies in detail exactly what an applet is allowed to do based on whether it is signed and who signed it. In this last set-up the TCB is both the VM and the code that reads the policy file and takes decisions based on it.

This development illustrates well the classic dilemma between flexibility and security: by making the security flexible and more user-driven we do allow adaptation to many application scenarios, but at the same time we also give user and system designers more freedom to shoot themselves in the foot! For instance, note that the security policy is just a file on the system, and it is not hard to imagine the consequences of a policy that gives an applet write-access to the directory that holds the security policy.
For details, read **Read Li Gong, p. 21-59.**

15.7 Exercises

Exercise 15.1 The security model which for Java (JDK 1.0) describes the relation between Java applets and permanently installed applications is called the sandbox model.

1. A multi-level security model is one where parts of the system are hierarchically ordered, from "high" to "low", where "high" could mean high confidentiality or high importance, for instance. A multilateral model is one where the the parts of the system are on the same level, but where you still want to control the information flow between different parts.

   Is the sandbox model a multi-level or a multi-lateral security model?
2. Does it use mandatory or discretionary access control?
3. Which security mechanism is used in JDK 1.1 and onward to make the sandbox model more flexible?
4. Which change from JDK 1.1 to 1.2 allows a more flexible version of the sandbox model?

This exercise is about identifying the security mechanisms used by Java to implement security policies:

- point out mechanisms based on cryptology and specify which aspects of the security policy they can support.
- point out mechanisms not based on cryptology and specify which aspects of the security policy they can support.

Exercise 15.3 Some models for security policies focus on either confidentiality or authenticity. Place Bell-Lapadula, Biba, Chinese Wall and Sandboxing in this table (see exercise 5.1 for definition of multi-level and multi-lateral):

	Multi-level	Multi-lateral
Confidentiality		
Authenticity		

Exercise 15.4 Give some examples where the following models for security policies are used. This can be non-it cases, similar to the functional separation example from the note, but can also be "electronic" cases:

- Bell-Lapadula
- Biba
- Lattice
- Chinese Wall
- Prevent-detect-recover
- Dual Control
- Functional Separation
- Mandatory and Discretionary access control.

Exercise 15.5 Does UNIX use

- Multilevel or multilateral security?
- Mandatory or Discretionary access control?
- Possibly mention other models used in UNIX

Exercise 15.6 Which model for security policy would you use in a system *a la* the Java sandbox model, where confidentiality is unimportant, i.e., we do not care if the applets read data outside the sandbox, we just want to ensure authenticity of those data, that is

- User processes cannot read data in the sandbox, but applets can read user data.
- User processes can write data in the sandbox, and applets can only write in the sandbox.

Exercise 15.7 Repeat the previous exercise, now with focus on confidentiality. Formulate two rules a la the above that protect the users' privacy against applets but ignores authenticity, and explain which model you would use.

Consider a secret service where users and information is classified in levels: public, classified and top-secret. Furthermore, information and users on the "classified" level are partitioned in to geographical areas: Asia, Europe and America. We do not want information to flow between different geographical areas, and information must not flow to a level that is less confidential. Draw a diagram showing a lattice that is consistent with these rules.

Exercise 15.9 Show that a lattice $(S, \leq)$, where S is a finite set, must have a "largest element", i.e., an element z such that $a \leq z$ for all $a \in S$. Note that an induction proof may seem like a good idea but this will not work very well. Instead you can start from an arbitrary element in S. Either this one is the largest element, or if not you can use the least upper bound rule to find a larger element.

Exercise 15.10 Consider any lattice $(S, \leq)$ and a finite set $A \subset S$ with at least 2 elements. Show that there exists a least upper bound for A, that is, an element $c \in S$ such that $a \leq c$ for all $a \in A$, and for any c' with this property, we have $c \leq c'$. Note that the definition of a lattice immediately says that such a c exists if A has one or two elements, so your task is to generalize this.

Exercise 15.11 At

```
https://www.nemid.nu/dk-da/om-nemid/historien_om_nemid/
oces-standarden/oces-certifikatpolitikker/POCES_Certifikatpolitik_version_4_Eng.pdf
```

you can read the security policy for OCES personal certificates (copy into your browser as one line). In chapter 7.2 it is described how to handle cryptographic keys. Do you think the policy is reasonable – is anything missing? Which of the basic models you have seen are used in this security policy?

Exercise 15.12 Suppose you are commissioned to construct a system preventing cheating at written exams, where people are allowed to bring their own laptops. Via a local area network, all machines get access to the same printer, which is to be used by everyone participating in the exam.

- Which threats would you include in the threat model?
- Which model(s) would you use for the security policy?
- Sketch a concrete security policy, i.e., what are the basic security objectives you think the system should satisfy.
- Mention some security mechanisms you would use to implement your policy. Note that they do not all have to be "electronic" mechanisms.

Exercise 15.13 (Electronic Election 1) *Danmarksdebatten* was a web site set up by the Ministry of Science. Here, public institutions could put various matters up for public debate. The site worked as most debate sites, but with a somewhat more complicated administration, because one has to keep track of a lot of different users who can start and moderate their own debates, but not those of others.

In 2005 it was decided to expand the functionality so it becomes possible to have non-binding voting via the site. The idea is that public institutions can define their own subjects to vote on. To vote, one has to be a user of the public digital signature system and one may have to satisfy certain extra demands defined by the party who sets up the vote. For instance, a city council can set up a vote on a local subject and decide that only citizens of the city can vote. All votings are non-binding, so the city council is not obliged to follow the result.

Voting takes place over the net, and the result is published on the site as soon as the deadline for the voting is over.

A company was commissioned by the ministry to implement the system. The company then presented the following set of security objectives to one of the authors of this book for evaluation:

- All voters who are qualified to vote can vote, but only those, and you can only vote once.
- After you have voted, you will be able to see the tally of votes cast so far.
- Only the final result must become publicly known, in particular it must not be known how a single voter voted. However, it is part of the purpose of the system to gather various demographic and statistical data that can be given to others. For instance, it can become known how many women voted yes. This has been verified to be in accordance with current law, as long as every group for which separate information is given, consists of at least 25 people.

Your task: it is quite clear that this pice of text is not a security policy as defined in the course(Def. 1 in Chapter 5). For instance, there is no specification of which system we are protecting (presumably the system to protect is not the entire Internet!). Furthermore one or more natural and important security objectives is/are missing. Rewrite the text so that it can be called a security policy and such that the set of security objectives is more complete than before.

16

Attacks and Pitfalls (DRAFT)

Contents

16.1 Categorizing attacks

In this chapter, we look at various ways to categorize threats against systems, and we give some examples of common attacks and the mistakes that make them possible.

As discussed earlier, any reasonable approach to design of secure IT systems must contain a *threat model*, specifying which attacks we are going to protect against - but also (perhaps implicitly) those we will not address. In order to have some assurance that we have thought about everything that is relevant, it can be very useful to have a number of ways to categorize attacks into different types. We start by looking at a number of such taxonomies.

16.1.1 What is the attackers goal: STRIDE

This is a way to categorize attacks according to their effect, or in words, what the attacker is trying achieve:

S poofing Identity: results in the attacker being able to impersonate another person (user).

T ampering: results in attacker being able to manipulate data without this being detected.

R epudiation: results in the attacker being able to deny having done something he actually did.

I nformation Disclosure: results in the attacker being able to access data he should not see.

D enial of service: results in the attacker being able to deny others access to a system.

E levation of privilege: results in the attacker being able to have more rights in a system than he was supposed to have.

The same attack may have several different effects at the same time, depending on the context. A man-in-the-middle attack, such as the one on the Needham-Schroeder protocol, usually results in spoofing of identity, but as a result, you may get information disclosure as well.

16.1.2 What means are used in the attack: X.800

One may also categorize attacks according to the *means* utilized by the attacker. The classification below is adapted from the X.800 standard published by the CCITT organization who is also behind the X.509 standard for certificates. The X.800 standard operates with two main types of attacks on network transmission, namely passive and active attacks. These can be further divided as follows:

Passive attacks:

- **Eavesdropping:** The attacker listens in and looks the information sent.
- **Traffic Analysis:** The attacker only looks at who is sending to who and how much is sent.

Active Attacks:

- **Replay:** The attacker resends an old message.
- **Blocking:** The attacker stops a message from arriving.
- **Modification:** The attacker changes the data sent or injects a new message of his own.

Clearly, passive attacks are very difficult to detect, so the obvious defense is to use encryption against eavesdropping. This cannot in itself stop traffic analysis. For this, a number of tricks are available, such as sending the same message to many parties to hide the intended receiver, or to deliberately randomize the length of messages to hide the size of the actual message. A final method is known as *steganography*, which roughly speaking means information hiding, i.e., methods to hide information inside a message that otherwise seems to have a different content. One example of this is to make slight changes to a digital picture, such that some of the bits representing the picture encode the message you want to send. However, if one chooses carefully where to place these message bits, the change is not easily visible. In this way, one can hide even the fact that a message is sent.

Active attacks are easier to detect, but harder to prevent. So focus is often on detection of attacks, using message authentication codes, signatures, sequence numbering of messages, etc. X.800 focuses on data transmission over networks, but the types of attacks also make sense for storage of data. Nevertheless, X.800 is somewhat limited in scope, and one should not think of it as a complete list of tricks an attacker can use.

There is a certain overlap between X.800 and STRIDE, which is not surprising as there is often a direct correspondence between the goal of an attack and the means used.

16.1.3 Where are we attacked, and by whom: EINOO

When analyzing the possible consequences of attacks, and deciding which kinds of attacks we want to protect against, it is often useful to think about who is attacking, and where exactly the attack takes place. First, we look at *who* attacks us:

E xternal attackers, who are not legal users of our system; or

I nsiders, who are registered as users of our system and so have some amount of access to start with.

Likewise, one can distinguish according to *where* the attacker is able to hit us. One talks about

N etwork attacks, where the adversary can only listen to and perhaps modify network traffic, that is, he is limited to using the means described in X.800. It is usually impossible to prevent that network attacks are attempted, but most of them can be defended against against using cryptography (beware of traffic analysis, however!).

O ff-line attacks, where the adversary gets unauthorized access to information that is stored permanently in our system, on disk or on other media. He can then steal and/or modify this information. Examples include stealing the password database file, or copying from a PC the file containing the owners private key for netbanking. Off-line attacks are harder to carry out than network attacks because they require breaking the system's access control to some extent, but on the other hand they can be attempted any time that is convenient for the attacker, and once access is obtained he can usually use standard search tools to find what he is after. Cryptographic protection of data is a obvious defense against off-line attacks, but one must realize that the keys used for this have to be stored somewhere, and if the off-line attacks gives access to the keys, this protection is useless.

O n-line attacks, where the adversary breaks into the system and observes it while programs handling sensitive information are running. He may now be able to read secret keys from the RAM, or modify what is shown on displays to users, e.g., to make them believe they are signing a transaction that is different from the one actually signed. On-line attacks are harder to mount than off-line: the adversary needs to completely break the system's access control and he must be there at the right time when the process he targets is actually running. He must also have sufficient knowledge of how the memory is organized to find what he is after. On-line attacks can have very severe consequences and can only be defended against by having different parts of the system protected in different ways so that one can reasonably assume that not all parts of the system are subject to the attack.

The EINOO classification can be useful when designing threat models for concrete systems. An example: suppose we are designing a system where a web server should collect sensitive data from some users and store them permanently. Which attacks should we worry about? if the server is hosted by a reliable organization using a well maintained machine and operating system, we might choose to say that we will worry only about attacks from outsiders, and only take care of network and off-line attacks. The rationale here is that since we trust the organization that hosts our server, and have sufficient confidence in the protection they have put up (firewall, intrusion detection etc.), we make the assumption that outsiders will not be able to do an on-line attack. There is of course no guarantee that such an assumption is actually true. But since it is never practical to protect against ALL attacks, some assumption always has to be made, and being clear and conscious about the assumptions you make is clearly much better than not knowing exactly what kind of security guarantees you have.

16.1.4 Where did we make the mistake: TPM

A final way to categorize attacks is according to what makes the attack possible, put another way, according to the way in which attacks exploit mistakes in our framework:

T hreat model: the attack is possible because the threat model was not complete: an attack we did not think of turned out to be relevant, or an attack we dismissed as being too unlikely turned out to be relevant after all.

P olicy: the attack is possible because the specification of the security policy expresses something different from what we intended. For instance because the policy was too complicated and we did not realize what it actually allowed.

M echanism: the attack is possible because the security mechanisms can be circumvented

The hope is that by being conscious about the TPM classification while studying attacks that happened to existing systems, one will know better where to focus when designing new systems.

16.2 Examples of Attacks

In the following a number of very different attacks are shown. This hopefully illustrates that identifying all relevant attacks is often extremely difficult. Hopefully, the list can serve as inspiration to specify a good threat model, but it is also intended to warn the reader to always be skeptical about the completeness of any threat model – perhaps particularly those you have designed yourself!

Apart from the attacks explicitly mentioned here, it is of course useful to recall all the different attacks that were already mentioned in the notes and exercises on cryptography, network protocols, etc.

16.2.1 Illegal Inputs

Perhaps the most important type of attack in practice is the type that circumvents security mechanisms by exploiting mistakes in the software that supposedly implements the mechanisms. This takes an amazing number of different forms, but the basic idea is always the same: The attacker gives some input to a program, where the input has a form that the programmer did not expect. This can cause the program to behave in some inappropriate way that the attacker may be able to exploit. This usually happens because the data is interpreted as something else than just data, say as a pointer or as code that can be executed.

Overflow Attacks

In an overflow attack, the adversary gives an input to a program that is longer than expected. If the programmer did not check for this properly, a multitude of bad things can happen. Programs written in C may have this problem, because C does not have any built-in protection against many kinds of run-time errors such as writing outside an array. On the other hand, C is very machine-near and fast and this is why lots of low-level software OS kernels etc. are written in C.

Consider the following simple procedure:

```
void foo(char* input){
   char buf[3];
   strcpy(buf, input);
}
```

The procedure foo declares a local variable, an array buf with room for 3 characters. It then copies the string pointed to by input into buf. In C, strings are 0-terminated and therefore the standard function strcpy will continue to copy until it sees a 0-character in the array it copies from.

This of course means that if the input string is longer than what we have room for in buf, the remaining characters will be written outside the memory that was allocated for the array. Namely, the bytes actually allocated for buf will be at addresses $\mathsf{buf}, \mathsf{buf} + 1, \mathsf{buf} + 2$, and the "overflow" characters from input will be written at addresses $\mathsf{buf} + 3, \mathsf{buf} + 4, \ldots$.

This can be bad enough in that the program may crash if this happens, but the real problem occurs if the input string can be chosen by an adversary. This could well be the case if the procedure foo was part of a web server, and is called to handle a form that a user has to fill in. An adversarial user might notice that the web server seems to have problems if he types a very long string into a field and may then start to wonder if this bug can be exploited.

To see why an attack may be possible, we have to understand what happens on the machine when the procedure foo is executed. Usually, the machine works with a *stack*, which on its top contains a *stack frame* which is a piece of data that is assigned to the procedure that is currently being executed. So when we call foo, this instance of foo is assigned a stack frame that is put on top of the stack. The stack frame has space inside it for all local variables of the procedure, 3 bytes for buf in this case. But is also contains the *return address*: if the program counter was at address addr when we call foo, then we should return to address $\mathsf{addr} + 1$ when foo is done. So the return address $\mathsf{addr} + 1$ is put into a fixed position in the stack frame. This means we have a stack frame containing 3 bytes for buf and the return address, and they will usually be placed next to, or close to each other.

This in turn implies that if an adversarially chosen string that is too long is copied into buf, then not only will the return address be overwritten, *it will be replaced by a value that the adversary can choose*. So this means that after foo is done, the machine will go to a place in memory decided by the adversary and start to execute whatever it finds there.

It should be obvious that this is a very dangerous scenario, in particular because a clever adversary might even be able to make the return address point to the very string he just supplied as input, which means that he gets to write the code that will be executed after foo. This code will execute with the same privileges as the attacked program so the adversary may now have access to all sorts of interesting stuff.

This type of attack exists in an endless number of variants and even "innocent" looking bugs can be exploitable in surprising ways. So,

it is safe to say that no bug is ever safe!

While an obvious solution seems to be to use a "nicer" language than C with protection against overflow bugs, this is often unrealistic for efficiency reasons. Instead programmers should become better at writing secure code. For instance, libraries exist containing safe procedures for string handling in C. Sadly, the main obstacle surprisingly often is that they first need to understand the problem and care about it.
For more details, read **Howard, p.127–138, p.155-170**.

The Unicode Exploit

Another set of examples are about problems with the IIS, the Microsoft web server application. One example of the type of problems it has, is the so called Unicode Exploit, where the idea is that the IIS is supposed to check whether the requests it gets from the outside try to access directories that should not be accessed. At the same time, requests are allowed to contain unicode characters which is a way to encode special characters like ø, å using only the "normal" character set. This allows directory names to contain all kinds of international characters. These characters must be decoded before taking action. Unfortunately, the IIS (or at least a previous version of it) does the security check of the requests *before* the decoding, or at least before all decoding steps are completed, so this allows an attacker to mask an offending request by encoding it in unicode in a special way so it takes a form that the security check will not recognize. The attack therefore consists of sending such a carefully masked request to the IIS, a request that will execute, e.g., a command shell. This will (or at least previously it would) run with the same access rights as the IIS, so the attacker can now upload and run various interesting software to the target computer.

The details of how a request was actually masked in order to fool the security check can be found in the text by McClure et al. mentioned below.
Read **McClure *et al.*, p.175–184**.

Cross-Site Scripting

In cross-site scripting, the attacker exploits poor design of various web pages. We describe here a very simple form of the attack which actually is automatically stopped by newer versions of most browsers. However, many other variants exist that are not so easily caught.

A web site might be vulnerable to cross-site scripting if it has a page that accepts input from a client and then eccoes this input back to the user in a new page.

As a typical example consider web site *naive.com*, which has a search form on one of its pages. The expected use is that a user fills in the form, say with a string "cross-site scripting" to search for, and presses GO. Next, the site shows a page where it says

```
You searched for: cross-site scripting
Search results:
  xx
  yy
  ..
```

While this may seem innocent, let's have a closer look at how this is actually implemented: typically, when the search request comes in, a new page is dynamically created. The page contains the search results, and also the string that was searched for, put in just after "You searched for:", In some cases, this is done by simply copying directly what was in the search box.

This turns out to be a very bad idea. The first problem is that it is possible to fill into the search box a string which is executable java-script code. When this string comes back to the browser in the result page, the browser will usually execute this code. Of course, this was not the intention, but the browser has no way to know what the intention was!

One could argue that this is the client's own problem: he should know better than to search for an executable string. But its gets much worse: when *naive.com* returns a result page, there is no guarantee whatsoever that the content of the "You searched for"-field actually comes from the client it is sent to!

Suppose the site *evil.org* has a page where it says "search naive.com" and then a link you can click – plus of course some plausible explanation why searching *naive.com* would be a great idea. If unfortunate user John Doe clicks the link, the URL which is sent to *naive.com* contains a search request, but as (part of) the string to search for, it contains a piece off java-script code designed by *evil.org*. When *naive.com* returns the search result, this code is executed by John Doe's browser!

Note that from John's browser's point of view, the java-script code comes from *naive.com*, a site John may know well and has said he trusts. So there seems to be no reason not to execute the code. Worse, since the code seems to come from *naive.com* it has access to this site's data. For instance, if John has an account at *naive.com*, his browser may store a cookie containing John's username and password to the site. So now, the code from *evil.com* is free to read this cookie and send the data somewhere else. This can be done, for instance by the code sending a http request to a URL controlled by *evil.org*, and include the data in this request.

Cross-site scripting is a particularly nasty threat because the place where the vulnerability is, is not the place that suffers the consequences. As a user, there is very little you can do, except perhaps turning of java-script when you are browsing suspicious places. Browsers can be designed, and are being designed to partially solve the problem. For instance, the simplest form of the attack that we described here (where code is embedded directly in a URL) is often stopped by modern browsers, they simply filter out code that appears in a URL. But the browser can never completely solve the problem. After all, it is supposed to receive (javascript) code and execute it and we cannot expect it to filter out all adversarial code. To do this, the browser would have to understand hat the intention is with each web site you visit, and this is of course not realistic.

The right solution is to stop the problem at the place it is coming from, namely web site designers should verify data they get as input more carefully, to make sure that code from other sites does not make into pages they send to their users.

The Heart Bleed Bug

OpenSSL is an open-source software package that implements SSL/TLS. It is used by a very large number of web servers across the world, including some of the big players such as Facebook and Google. Despite this, OpenSSL was maintained by a rather small group of volunteers, at least up to 2014, where one person was permanently employed. It is therefore maybe not surprising that bugs occur, sometimes with serious consequences.

In SSL/TLS one can have a so-called heartbeat, which means that the client can confirm regularly that the server is still alive. This is done by sending a nonce (in this case a character string) to the server and have the sever return the same nonce. The client sends both the nonce and its length. As a result the server has at some point stored in RAM both the nonce (in an array) and the length ℓ. It will then produce the message to send the client by reading ℓ characters from the array.

The Heart Bleed bug was a simple case of missing input validation: in an update of the software, it was forgotten to verify whether the length sent by the client matches the actual length of the nonce. Now, if a malicious client sends a length that is much too large, the server will continue to read beyond the end of the array and will return some piece of its internal memory to the client. This is potentially a serious problem because the memory may contain confidential data, such as the server's private decryption or signing key.

The bug was discovered in the spring of 2014, and had existed for about 2 years. It potentially affected a large fraction of the servers worldwide, but there are no reports confirming that any criminals exploited it. It may well have been known to various secret agencies such as NSA, but this is of course is not confirmed either.

16.2.2 IBM 4758

This is an interesting example of a case where the security policy was incorrectly specified. In the case of this hardware unit, the security mechanism is formed by the hardware protection of the box, which prevents an attacker from reading data in the box and from modifying the code it runs. No one knows how to circumvent this mechanism.

The security policy is formed by the API offered by the box, as this specifies which operations an outsider is allowed to do to the data held inside the box. Via an attack found by some PhD students from Cambridge, it turns out to be possible to (mis)use the API and efficiently extract cryptographic keys from the box without having permission to do so (or at least it used to be possible — IBM has changed the API since then). This was done by simply using the functions in the API as they were defined, although in a different sequence than the designers had expected. So this is an example of an attack exploiting that the security policy was not specified as intended.

The main reason why the attack was possible is that the 4758 does both single DES encryption with a 56 bit key, and two-key triple DES where two DES keys are concatenated to form a 112 bit key. Single DES is not considered secure anymore and is mainly there for backwards compatibility, but this actually constitutes a security risk, as we shall see. Skipping many technical details, the essence is as follows:

A first observation is that the attacker can have the box choose a 56 bit DES key K_0 and encrypt some given data under K_0. There is no way to have the box give you K_0 directly, but given the matching plain and ciphertext, one can find K_0 quite efficiently by exhaustive search, a 56 bit key is simply not long enough these days.

So the real question is if we can steal some of the *double length* keys the box holds?

To this end, we need to know the way in which one can transport a (double length) key from one 4758 to another. Let K be such a key, held in one box. One can then ask it to split the key in two *key parts*, K_1, K_2 where $K = K_1 \oplus K_2$. The idea is now that the key parts are sent to the other side via two different channels, say, transported by two different persons, who then manually key in their respective key parts. This allows the receiving box to compute the key, one can ask it to do so by calling a function called *Combine Key Parts*. Further, once the key K has been computed, and if it has been defined to be a so called key encryption key, one thing you can ask for is to use K to do an *Export Key* operation which will make the box output $E_K(K')$ where K' can be virtually any other key stored in the box. Hence, if a user of the box can get into a situation where he knows both K_1 and K_2, he can steal all keys from the box very easily.

Of course, this could happen during a normal transport of a key, if the two users who know the key parts would collude. But there is nothing new in this, this is an already recognized risk. The interesting thing is that a user can do an attack without colluding with anyone, assuming only that he has permission to do the Combine key parts operation.

To this end, one can exploit the unfortunate fact that there is (was) no complete separation between the use of single and double length keys. For instance, it is possible to call a function that forms a double length key by concatenating a single length key with itself, a so called *replicated key*. This function is there for backwards compatibility, since using a double length key where both halves equal single key K is equivalent to single DES encryption under K. Of course, the designers knew that such a key is not very secure, so you can only export single length keys using such a key. However, the attacker can do as follows:

1. Have the box create a single length key K_0 internally and compute K_0 using exhaustive search, that is, ask the box to encrypt some known data M under K_0 and use the resulting ciphertext C to test all possible keys until the correct key that encrypts M to C is found.
2. Ask the box to form a random double length key encryption key K_2 with *replicated* halves. Ask the box to export K_0 protected under K_2. This is legal since K_0 is a single length data key so it is not stronger than K_2.
3. The above step produced $E_{K_2}(K_0)$. We can now find K_2 by exhaustive search as above. Note that we have to define K_2 as a key encryption key that can be used for exporting other keys, for the purposes of the rest of the attack. This also means that it is illegal to use K_2 for encrypting data. Hence the detour where we first make a known single length key.
4. Manually type in a *non-replicated* key part K_1, pretending you are doing a transport of a key.
5. Call Combine key parts to produce $K = K_1 \oplus K_2$
6. Becuase K_1 is not replicated, neither is K, so it will be classified as full strength key encryption key. Moreover, it is not a key part known by a single user. You can therefore export under K all the keys you want from the box. Nevertheless, the attacker knows K.

The attack is also an illustration of the dangers of having different levels of security side by side in the same system, here represented by single and double length DES keys. If you can make the system mix the two levels, you can often do a successful attack.

16.2.3 GSM

This is perhaps not really a concrete attack, rather it is an illustration of a series of weaknesses caused by an inappropriate use of cryptography. A primary problem has been that the designers of the encryption algorithm used insisted on keeping it secret, thus violating Kerkhoff's principle that one has to assume that the algorithm will become known at some point. Sure enough, the algorithms were reversed engineered and published before long. They contained several problems that would surely have been found if the algorithms had been published in advance.

For details read **Anderson, p.352-363**.

16.2.4 Covert Channels and Side Channels

Taking as point of departure the Bell-LaPadula (BP) model, we find a type of attack which is typically done by insiders in order to foil the security policy, by circumventing the trusted computing base that was supposed to protect the system from its users. According to BP, information must not be sent "down" from a higher to a lower security level. But if we imagine a system where the levels share a common harddisk, it may be possible for a user with access to a high-security file to signal its contents to users on lower levels. One way to do this could be to move the reading head of the disk at certain pre-agreed times, say outside the actual disk if the next bit is 1 and somewhere inside otherwise. This will affect the time it takes before the next read request to the disk gives an answer, and some user on a lower security level may be able to observe this. This phenomenon is called a covert channel and can be very difficult to prevent if different parts of a system share the same hardware.

A very large number of other channels have been found, this type is also sometimes known as *side channel attacks*, in which the scenario is that an external adversary is trying to steal information from the system using a "channel" that the system designers were not aware of. This leads to an almost endless series of scenarios where the threat model turns out to be incomplete.

One example of this is the attack we discussed in the chaper on Key Management and Infrastructures where you could steal a secret key from a chip card by measuring its power consumption while it was using the key. Here is another example:

When using CBC encryption, the final block must be padded in some way, so you have an integral number of blocks. Certain protocols instruct the receiver to check after decryption if the padding has the right form and send a message to the sender saying whether an error was detected. Now, consider an attacker who has seen some encrypted text C. The attacker is now free to fabricate from C some manipulated ciphertext C', send it to the receiver, and observe whether he sends back "OK" or not. The important point is that with certain padding schemes, if C' was sufficiently cleverly constructed, the attacker will be able to find information on the original plaintext after a few attempts. It has been demonstrated that in this way, an encrypted password can be found efficiently from real transmissions. We will not describe the technical details in the construction of C', since this is not the main point in our discussion.

This attack does not work when using SSL/TLS, even if CBC and the "dangerous" padding is used. The reason is that *everything* sent is encrypted, including error messages, so the attacker cannot get the information he needs. So everything should be OK?

However, Canvel *et al.* at the Crypto 2003 conference found that there are still problems. The attacker can instead measure the time it takes the receiver to answer. If a padding error is found, we can expect an error message to be sent almost immediately. But if there is no padding error, the server has to go on according to the protocol and check a MAC before the final OK can be sent. This will take more time, and if the time difference can be observed from the outside, the attacker has essentially the same information as before, and the attack still works. Canvel *et al.* tested their attack on real networks and found that if they listen from a point sufficiently close to the server (in terms of the number of nodes between them and the server), the attack indeed works. The attack was found to work with two switches and a firewall between server and attacker. As one might expect, the information needed disappears in random networks delays if they move too far away.

16.2.5 The Xbox

The Microsoft game console Xbox is physically constructed very much like a normal PC. But of course, the manufacturers do not want users to use the console as a standard machine, since this opens up possibilities of playing copied CD's and DVD's and all sorts of other unwanted ways to use the machine.

In the first version of the Xbox 360, some measures have been taken to try to ensure that the box always boots up in a preset state (from where it will only do things it is allowed to do). There is of course a boot record in the system that would normally contain the very first code that is executed when booting.

However, in 2005, a PhD student from MIT (Andrew "bunnie" Huang) found out that this visible boot record is a decoy. In fact, the box contains a secured hardware unit which stores the real boot record. At boot time, the real thing is copied to the right location via a high-speed bus. This real boot record contains a secret key and code that is used to decrypt and verify some further code that is stored in external flash memory. This code, once decrypted and verified, forms the kernel of the operating system.

The real boot record is transferred to the CPU in clear text, and apparently this was assumed to be secure because of the high speed of the bus. However, Huang showed that it is possible to read the key from the bus using fairly standard equipment. Of course, once you know the key, it is possible to replace the code for the system kernel by something else. An important remark here is that the secret key is the same in *all* Xboxes! This may be a good choice in terms of making the manufacturing process easier and cheaper, but is of course a serious security problem.

There are more layers of security in the Xbox, so this hack alone does not allow to run arbitrary applications on the Xbox. However, it may well be that someone found a way around all of the security measures.

16.2.6 A Backdoored Pseudorandom Generator

This is the story of how, allegedly, an American three-letter agency has interfered with the standardization process of a cryptographic algorithm by NIST. As a result of this, the agency might have been able to bypass the encryption layer of the SSL/TLS protocol, thus facilitating surveillance of internet traffic. Thus this is an example of how, sometimes, even a very paranoid threat model might not be paranoid enough.

Psuedorandom Generators

To understand the story one needs to know a little bit about *pseudo-random generators* or PRGs for short. Good randomness is an essential tool of any system using cryptography. When a user generates any secret in a cryptographic protocol, the process must be unpredictable from an outsider point of view. Consider for instance a silly example in which a user chooses their secret key to be equal to the time of the day. Then an adversary can easily predict what the value of the this key is, and would be and therefore break the security of the the encryption scheme. Unfortunately, good randomness is hard to come by and expensive. While *true randomness* generators exists (for instance, devices that measure unpredictable and complex physical phenomena), such devices are quite expensive and therefore most PCs and other consumer grade hardware are not equipped with them. Regular PCs collects all the “entropy” they can from unpredictable sources such as you typing on the keyboard, the movement of your mouse, etc. This generates few bits of “true randomness” which are unfortunately not enough for the need of a modern computer system.

Therefore, most computer systems use PRGs to “expand” the few bits of true randomness they have available to strings of unbounded length which looks random, in a way similar to what has been described in Section 5.1.6 when talking about stream ciphers (in fact, one could use a good stream cipher, such as a block cipher “in counter mode” to construct a good PRG).

We will introduce a little notation for working with PRGs. A PRG is an algorithm G that on input the current state s_i outputs some random looking string r_i and the next state s_{i+1} in the following way

$$(r_i, s_{i+1}) = G(s_i)$$

The PRG is initialized with some truly random string s_0 known as the "seed". Note that there is nothing random about a PRG after the seed has been fixed, in other words, given a seed s_0 then the entire output of the PRG is fully determined. Moreover, given any state s_i it is possible to completely predict the future outputs of the PRG and therefore the state should stay secret. On the other hand, given access only to the output strings r_i, it should not be possible for any (computationally bounded) adversary to predict the next PRG outputs (or in fact, distinguish the r_i's from truly random strings).

Dual EC DBRG

In 2007 the US standardization body NIST standardized a PRG called "Dual EC DBRG". This PRG uses elliptic curve cryptography, which is beyond what you have learned in this book. However, it is possible (by forgetting many details and oversimplifying a bit) to explain the main ideas using only the math which you have seen in Section 14.4.1 when describing the Diffie-Hellman Key Exchange.

In a nutshell, this PRG uses two (hardwired) "points on an elliptic curve" P and Q, which in our simplified explanation will just be two large numbers between 1 and some prime number n The PRG is specified as following:

1. Take as input the current state s_i (which is a number between 1 and n);
2. Compute the next state as $s_{i+1} = P^{s_i} \bmod n$;
3. Compute the output as $r_i = Q^{s_i} \bmod n$;

It is possible to give a mathematical proof that, if P and Q have been choosen randomly and are large enough (2000 – 3000 bits), without any influence by the adversary, then this is a secure PRG. However it is also very easy to see that, if the adversary gets to choose P and Q, then it is very easy to break the PRG. This was no secret, and was immediately pointed out by some Microsoft researcher at the CRYPTO conference in 2007, shortly after the NIST announcement that they were going to standardize this PRG. Note that at the time no one had the fantasy to assume that this was being done with malicious intent, and it was just assumed that NIST had made an honest mistake. However the standardization process continued, and eventually the PRG was standardized. Crucially, the standard also contained two predetermined values of P and Q that users where invited to adopt. No discussion on how P and Q where chosen was provided in the standard.

Let's now see how it is possible to break the PRG for an adversary that has chosen P and Q in a malicious way.

1. First, the adversary chooses a random number Q, and another random number x. Then, the adversary computes $P = Q^x \bmod n$. The adversary gives P, Q to the unsuspecting user, and keeps x for himself.
2. Then, we assume that the adversary observes some output of the PRG r_i. We will return later on whether this is a realistic assumption. For the moment, imagine that the PRG is used in an eletronic casino, as the source for determining the next number which should be output by the roulette. Therefore, the output of the PRG can be observed by anyone.

3. Finally, the adversary computes the next state $s_{i+1} = (r_i)^x \bmod n$ and can therefore predict all future outputs of the PRG (and, in the online casino example, become filthy rich!)

To see why the attack works we are going to use the commutative property of exponentiations as we have done in Section 14.4.1.

Remember that the rule for updating the state of the PRG is $s_{i+1} = P^{s_i} \bmod n$. But the adversary chose P as $Q^x \bmod n$ so we can write $s_{i+1} = (Q^x \bmod n)^{s_i} \bmod n$. Using the commutativity of the exponentiation we can then write $s_{i+1} = (Q^{s_i} \bmod n)^x \bmod n$. If we now remember that the rule to compute the output of the PRG is $r_i = Q^{s_i} \bmod n$, we can conclude that $s_{i+1} = (r_i)^x \bmod n$ and therefore the adversary has successfully managed to compute the next state of the PRG.

Breaking TLS with Dual EC DBRG

As discussed above, the attack only works if the adversary gets to see the "raw" output of the PRG. This is not always the case: For instance, if you use a PRG to generate a secret key, you are not going to publish the secret key!

To see how one can use the backdoor of Dual EC DBRG to break cryptography, remember how SSL works, as described in Section 14.3.1.

1. In Step 1 of the protocol, the client generates a random nonce n_C; This value is sent unencrypted to the server, so anyone eavesdropping on the channel will learn n_C;
2. In Step 3 of the protocol, the client generates a random pre master secret pms. This value is then encrypted under the public key of the server.
3. It is very important that the pms stays secret: the key used for data exchange is generated in Step 7, as a function of n_C, n_S (which are transferred in plaintext over the channel) and pms (which is the only unknown value from the adversary's point of view). Thus, if the adversary learns pms, then the adversary can compute the data exchange key and completely decrypt the entire SSL traffic.

At this point it is possible to see how the attack works: the adversary observes $n_C = r_i$, one of the outputs of the PRG. Using the backdoor x the adversary can compute the next state s_{i+1}, and therefore the adversary can predict the value of the next output by evaluating the PRG e.g., $(pms = r_{i+1}, s_{i+1}) = G(s_{i+1})$.

16.2.7 SPECTRE

In 2018, a very important new type of attacks emerged that in a very fundamental way challenges the way modern CPUs have been optimized for speed. In this section we will discuss the basic problem and sketch a simple example of the attack. It is not possible to give a complete overview: in particular SPECTRE is actually a family of related attacks, and the number of forms in which it exists is growing even at the time of writing.

To understand what happens, we need to first discuss the physical memory organization of a modern computer: as CPUs get faster, the process of fetching data from the main memory (RAM) relatively gets slower and slower. Therefore all modern CPUs have *caches*. A cache is a fast on-chip memory of limited size, and the basic idea is that when data are fetched from RAM, the CPU also puts it into the cache (and may also fetch other data that are stored close to the item in question). The hope is that, at least for a while, the CPU will only need data that are already in cache and hence are available fast. Of course, this will sometimes fail, in which case we have a so called *cache miss*.

In case of a cache miss, the CPU will have to wait until the data it asked for is available. Or does it? Enter the main issue we have to discuss: *speculative execution*. What most CPUs do is to make educated guesses at which instructions will have to be done after the data is available from RAM, and then execute them (if possible). For instance, the data we are waiting for may be controlling whether a conditional statement should be executed, but the CPU may guess that the statement should be done and just execute it. The CPU will store a copy of its registers before it starts the speculative execution so that if the guess was wrong and what it did was not supposed to happen, it can roll back to the previous state. This is no worse that if it had been idle while waiting for data. On the other hand, if the guess was correct, the CPU has saved time.

It is not uncommon to see speculative execution of several hundred instructions, and speculations on what will happen can be correct more than 80-90% of the time. This idea therefore has a major effect on performance.

So what's not to like about this? The basic problem is that even if the CPU rolls back to a previous state if it guessed wrong, some side effects of the speculative execution may not be erased by this action. In particular, if the cache has changed state as a result of the speculative actions, this is not reversed by the CPU registers being rolled back. This may have some very unfortunate effects, as we now explain by an example:

Suppose an adversary wants to steal data from some program while it is running. For instance, the adversary could be a user in a multi-user system and he is now trying to steal data from another user. The attack could also happen in a browser, where a malicious web-page is trying to steal data from another webpage that is running inside the same browser.

Suppose the target program contains the following small piece of code:

```
if (x < array1_size)
   y= array2[array1[x]*4096];
```

We assume the attacker knows the source code of the target program and that he can arrange for the following conditions to be true when the code above is about to be executed:

- The attacker controls the value of x, perhaps because x is an external input to the program.
- The CPU branch prediction believes that the check in the if-statement will evaluate to True. This could be the case if the adversary can make the target

program run the critical code several times with values for x that are in range, before doing the actual attack.

- x and array1 are in cache, while array2 and array1_size are not in cache (note that array1_size is a variable and not a constant). Again, if the adversary can give input to the target program, he may create this situation because the choice of input indirectly affects which data will be used more and hence which data are in cache.

If indeed we are in this situation, the attacker will get the code to execute with a value of x that is too large and hence array1[x] can be almost any byte in the memory that the target program has access to.

When we get to the critical piece of code, the CPU will start to fetch array1_size from the main memory, but it also will start executing the code inside the if-statement since it guesses that the condition will be true. $\mathsf{array1[x]} * 4096$ can be quickly evaluated since everything is in cache, so then the CPU will need to start fetching $\mathsf{array2[array1[x]} * 4096]$ to cache[1]. Now, while $\mathsf{array2[array1[x]} * 4096]$ is on its way to cache, array1_size arrives and the CPU realizes its mistake. The state of the registers is rolled back, but still:

$\mathsf{array2[array1[x]} * 4096]$ *is in cache, while no other element from* array2 *is in cache.*

Note that if the attacker can find out which element from array2 is in cache, this will immediately reveal the value of array1[x] which is the data the attacker wants. Of course, if the attacker could get the target program to execute any code he wants, this would be easy: just load every element from array2 into a register, one after the other, and measure the time taken for each element. One of these will load much more quickly than the others, namely the one that was already in cache.

Of course, the target program containing exactly such a piece of code is too much to hope for for the adversary. However, as we just demonstrated, it still true that the timing of various operations done by the program depends on the state of the cache. Moreover, the cache is a single piece of hardware that is used by all processes running on the CPU and so the timings of accesses to memory and cache done by different processes are not always independent. Based on this, several existing methods have been found for figuring out which data from a target program are in cache even if the adversary can only observe from a different process. We will not discuss the details further here but just give one simple alternative method: if the adversary knows the content of array1, then immediately after the attack we described, he can provoke execution of the same code, now with a value x′ that is in range, and time the execution. If $\mathsf{array1[x]} = \mathsf{array1[x']}$, this will be fast and otherwise it will be slow. Thus the adversary learns the value of array1[x] if the execution is fast. If the involved values are bytes, the adversary

[1] Scaling the address by the factor 4096 has (essentially) the effect only the relevant element from *array2* is fetched. This has to do with something called hardware pre-fetching which we will not discuss here. The adversary will have to hope that such a scaling is in the target program for other legitimate reasons.

can hope to learn the correct value with probability 1/256 in one go. This is not great, but certainly also not a negligible success probability.

It is important to realize that, while the SPECTRE family of attacks are not easy to exploit in practice, proof of concept code has been developed that demonstrate the attacks on many different CPUs, both in C and in Javascript running in browsers. Thus this line of attacks cannot be ignored although we do not yet know any such attack happening "in the wild".

The fundamental problem that SPECTRE points out, is that the speculative optimizations break the contract between software and hardware developers – in the sense that even if the software is safe in itself, there can still be serious security problems, because what the hardware does is not always exactly what was written in the program.

Even if the problem is serious, solving it is very hard. Simply turning off speculation is possible, but unacceptable in practice because of the serious performance penalty. There are some attempts to have compilers help solve the problem: the compiler will try to recognize critical code and turn off speculation temporarily while that code is running. This will certainly help, but is clearly only a partial solution. It would seem that the only way to really make SPECTRE go away is to develop fundamentally new types of of microarchitecture for CPUs.

16.3 Exercises

Exercise 16.1 Categorize the attacks from the notes on IIS, IBM4758 and CBC in SSL, using STRIDE, X.800 and TPM. Argue why you made the choices you did.

Exercise 16.2 Recall as many attacks as you can from the previous sections of the notes and exercises, such as Needham-Schroeder, and categorize what you find, according to STRIDE, X.800 and TPM.

Exercise 16.3 (Electronic Election 2) Consider the voting system for the web site "Danmarks Debatten", for which you studied a security policy last week. The system was actually been built as a prototype, and below follows a description of the solution that was proposed. The system consists of 3 main parts:

Server - the central hard- and software that collects and counts the votes cast.
Client - the hard- and software used by a user to cast his vote.
Database - the persistent memory used on the serverside to store information on the votes cast so far.

When a vote is cast, the communication flow is as follows:

1. At the entrance to the election, a secure SSL connection is set up with only server authentication, so the client authenticates the server, but the server so does not know which client it talks to.
2. To check that the client has a valid digital signature, a new temporary SSL connection *Temp* is set up with both server and client authentication. This

will fail unless the client has the secret key corresponding to its certificate. It is checked whether the user has voted before, and a lookup is done in the central CPR register (via a new SSL connection), where data on the voter is collected. This can be used to check whether the user is eligible to vote. For instance it may be required that you live in a certain area to be able to vote. If everything is OK, a token is handed to the client via $Temp$, namely an MD5 hash of session ID and the server's system time.

3. If the temporary session was successful, the original connection proceeds and gets the vote from the client. The client must also produce the token from the temporary session. This is to check that we here speak to the same client as in the temporary session.

In the database, the data stored for each vote cast is organized as follows: data is stored in two tables, $T1$ and $T2$. In $T1$ we have an entry for every vote cast in every election defined in the system. This entry contains a SHA1-hash of the CPR-number of the voter, concatenated with a secret key K. If you know K, this table allows to check whether a user with a given CPR number has already voted.

$T2$ also has an entry for every vote cast in every election. An entry contains the vote cast, and some data on the voter, namely age, sex and the area where the voter lives.

The system has a number of back-end users, who administrate elections. Of course, there are also security issues here, but we leave this issue out of scope in this exercise.

The system is planned to be hosted by a company who is obliged by contract to keep the operating system up to date with security updates. The host machine runs behind the company firewall. In addition, extra security measures are taken as follows: an additional firewall is running on the machine itself. An intrusion detection system called Tripwire is used. All systems files are checked regularly for integrity, and installed software is monitored daily for known security related bugs.

Now, the exercise is the following:

1. Put together a threat model for this system. Use, for instance, the EINOO classification. Which attacks might occur? Try to form an opinion on how likely the attacks are and how bad the consequences would be, should the attack succeed. For instance, is breaking into the database as serious as breaking into the server? does it matter *when* you break in? As a result, try to conclude which attacks it would be reasonable to protect against.
2. Study the solution sketched above, and compare to the security policy from last week and the attacks you just listed. Does the solution live up to the policy given the attacks you listed? if not, what would it take to repair the solution? Could the solution be simplified without affecting security?

Exercise 16.4 Try to come up with a covert channel type of attack on your solution to exercise 5.12 from last week.

Exercise 16.5 For people who have windows on their machine: on `http://www.microsoft.com/mspress/books/companion/5957.aspx`,
you can download under "Companion Content" the example programs mentioned in the text (as a Windows installer). Download the examples and run them to see for yourself the unwanted effects.

Which programs mentioned in the text are we refering to?

Exercise 16.6 This exercise looks at the example *StackOverrun* from Howard.

- Modify the code so the attack will fail.
- Is it possible to prevent the attack without changing the code? if so, how?
- How does Java protect against buffer overflows?

Exercise 16.7 In McClure *et al.* a buffer overflow attack on IIS is mentioned, using a program called jill. It is stated that the attack is hard to stop using a firewall. What could a firewall concretely do to stop the attack? What types of firewalls would be able to follow your suggestions?

Exercise 16.8 Explain how the Unicode-exploit from McClure et al. works. Imagine that the IIS was written in Java and was run inside the sandbox. Would the attack work now? explain why or why not.

Exercise 16.9 Give examples of attacks —and describe them in terms of STRIDE and X.800— that you can think of on the GSM system as described in Anderson.

Exercise 16.10 When you log into a website, the site may need to track you somehow. This is normally done by storing a *cookie* in your browser. A cookie is just a little piece of text which the browser will send back to the server on subsequent request.

To track you, the website will assign you a temporary session ID when you log in. The server stores the session ID in its own database and will also store it in a cookie in your browser. When you browse around on the site, your browser will present the session ID in the cookie, and the website will recognize you.

From the above, it should be clear that cookies often contain confidential information. For that reason, the browsers implement a *same-origin* security policy. The policy requires that a cookie is only sent back to the server it came from. So if I log into, say `http://paypal.com/` the cookie created by Paypal must never be sent to any other server.

In this exercise we will investigate how *cross-site scripting* (XSS) attacks can be used to circumvent this security policy. In these attacks one uses JavaScript to trick a webbrowser to send the cookie information to a third party site. When visiting a site one would normally not expect the JavaScript to do this, but if the page is insecure, an attacker will be able to inject untrusted JavaScript into a webpage. The exercise only looks at the most basic form of this attack where you embed code directly into a url. Many browsers will stop this type of attack by default. In most cases, however, you can switch off this protection and do the exercise anyway. Firefox has been known to be quite insecure for cross-site scripting and hence is good candidate browser for this exercise. At the time of writing, at least some versions of Firefox allows the attack directly.

We will use two pages in this exercise. The first page is a typical search form: `http://cs.au.dk/~mg/xss.php`. The other page is the attackers page which is used to collect the data: `http://cs.au.dk/~ivan/img.php`. Now work through the following bullits. You do not have to make an answer for the first one. For the rest, your answer should contain the line you typed into the browser to get the desired result.

- First go to `http://cs.au.dk/~mg/xss.php`. This is a typical search form. After searching for something, you'll be presented with a page that lists your query and any search results.

  Interact with the page and see how it behaves. What happens if you search for a string like `<i>hello</i>`, i.e., a string containing HTML?
- Make a search query which results in an image being shown in the search results. To include an image, you have to use HTML code that looks like this: `<img src="http://example.net/img.png">`.
- JavaScript is a client side programming language interpreted by browsers. It shares much of its syntax with Java, in particular, you escape and concatenate strings like in Java. Among other things, JavaScript allows you to add new content to a webpage. This will inject the string "Hello from JavaScript" into the HTML code:

  ```
  <script>
  document.write("Hello from JavaScript");
  </script>
  ```

  Combine the technique from the previous question with JavaScript, to make a piece of JavaScript that, when typed in the search form, makes the browser display an image.
- JavaScript has access to the cookie belonging to the website where it is executed. The cookie is a string in `document.cookie`. Make your query print out your `http://au.dk/` cookie.
- Assume that as an attacker you have made a website for collecting cookies from other web sites. You may do this my making the browser send a request to your server where the cookie from the other place is included in the request.

  As our attacker web site we will use `http://cs.au.dk/~ivan/img.php`. Like the search form, this page accepts a `q` parameter. The value of the `q` parameter will be rendered as an image. Construct a URL that will display an image with the text "Hello World".
- Combine the answers to the previous questions to do a successful XSS attack on the search form. The cookie should be rendered as an image on the site `http://cs.au.dk/~ivan/img.php` to demonstrate that you have violated the same origin security policy. Note that in javascript, you can assign a new value to `window.location` to redirect to another website.

17

Advanced Blockchain (DRAFT)

Contents

17.1 Dynamic Parameters

Above we assumed that important parameters like $\mathsf{Tickets}_i$ and Votes_i just were given. From a beginning probably they are: in the genesis block. But at the system evolves, so must the parameters.

As an example of the importance of evolving parameters, consider $\mathsf{Tickets}_i$. In a proof-of-stake system, your number of tickets is proportional to the holding on your account. So far the above system keeps using the same number of tickets that existed at genesis. This can be dangerous. Remember that we believe that most tickets will remain honest because the ticket holders have a lot of money in the system. But money move around. So it might happen at a later day that a majority of the ticket holders have no money in the system any more. That would leave them with no incentive to participate in the system or to do it honestly.

To avoid problems with moving stake, the system must occasionally update the number of tickets per account. It is important that all parties agree how many tickets each account has. To ensure this one can for instance let the tickets be the holdings of the accounts at the latest finalization point. When changing the parameters there can be a moment of "flickering" as not all parties change parameters at the same time. The tree layer is robust and can tolerate short moments of flickering. It is no worse than having a lot of corrupted parties for a moment that are using wrong parameters, and the tree layer can tolerate a lot of corruption if over long enough time there is honest majority. To avoid having to much flickering one can choose to only change parameters for instance each day. As usual there are many details to get right. We will now go into some of these.

17.1.1 Block Parameter

It turns out that it is convenient to require that dynamic parameters are a function of the paths in the tree, and not some protocols run on the side. Recall that it is important to agree on the changing parameters. Recall also that agreement is a hard problem. This is in fact the problem that the tree is trying to solve. It is therefore all but silly to not use the tree to keep track of the parameters.

Definition 17.1 (block parameters) We define a function $\mathsf{BlockParameters}$ which at each block in the tree specifies what are the system parameters as computed at that block. Let Block be a block in Tree. Let

$$\mathsf{Path} = \mathrm{PathTo}(\mathsf{Block})$$

be the path from genesis to Block in Tree. The function

$$\mathsf{BlockParameters}(\mathsf{Tree}, \mathsf{Block})$$

is required to only be a function of just

$$\mathsf{Path}$$

. In the following we will write $\mathsf{BlockParameters}(\mathsf{Block})$ for brevity. At each block it returns a structure

$$\mathsf{Params} = \mathsf{BlockParameters}(\mathsf{Block}) \ ,$$

which contain the following entries:

- Tickets
- FinalityCommittee
- Votes
- Hardness
- Seed
- SlotLength
- ...

The entry $\mathsf{Tickets}$ is a dictionary which for a public key vk of a party returns the number of tickers $\mathsf{Tickets}[\mathsf{vk}]$ of the party. The entry Votes is a dictionary which for a public key vk of a finalizer returns the number of votes $\mathsf{Votes}[\mathsf{vk}]$ of the finalizer. Note the these parameters depend on the path and therefore in particular can depend on the entire state of the overlaying smart contract layer, for instance the holdings of the accounts or some complicated protocol that was run to determine who has how many tickers and votes. For simplicity we can imagine that there runs a special designated smart contract on the blockchain, the parameters contract, which computes the new parameters. The function $\mathsf{BlockParameters}$ would then just read the state of this contract.

$\triangle$

17.1.2 Slot Parameter

The $\mathsf{block\ parameters}$ of a block are then just $\mathsf{BlockParameters}(\mathsf{Block})$. Having block parameters do not solve all the problems. Sometimes we want to know parameters associated with a *slot* slot. We for instance want to know the number of tickets a party had at that slot to see if it actually won the lottery for the slot. In general it is convenient to have parameters defined at each slot. If a slot has a block with that slot number, then it is easy. The parameters of the slot are the parameters of the block. If a slot does not have a block, then it is natural to tage the block right before the slot.

Definition 17.2 (slot parameters) We define a function $\mathsf{SlotParameters}$ which at each slot specifies what are the system parameters as computed at that slot. It takes three parameters

$$\mathsf{SlotParameters}(\mathsf{Tree}, \mathsf{Block}, \mathsf{slot}) \ .$$

It is require that Block has a slot number larger or equal to slot. Let

$$\mathsf{Path} = \mathrm{PathTo}(\mathsf{Tree}, \mathsf{Block}) \ .$$

Let $\mathsf{Block}_{\leq \mathsf{slot}}$ be the last block in Path with a block number that is $\leq \mathsf{slot}$. Then

$$\mathsf{SlotParameters}(\mathsf{Tree}, \mathsf{Block}, \mathsf{slot}) = \mathsf{BlockParameters}(\mathsf{Tree}, \mathsf{Block}_{\leq \mathsf{slot}}) \ .$$

We write $\mathsf{SlotParameters}(\mathsf{Block}, \mathsf{slot})$ for brevity.

△

17.1.3 Advance Slot Parameters

It turns out that having slot parameters is not enough either. Recall that in the lottery, I need to know whether I win some *future* slot. If I use the slot parameters of that slot, then the number of tickets and hardness of the lottery would be parameters of the block I would create if I won. This is a infinite definitional loop. We therefore want to make the lottery parameters for a future slot be the slot parameters of some earlier slot. In general it is good to know these parameters well in advance so one can know in advance whether one is the slot winner for a future slot, to get ready to quickly bake the block and send it. We make this advance time a parameter $\mathsf{AdvanceTime}$ of the system.

Definition 17.3 (advance parameters) We define a function

$$\mathsf{AdvanceParameters}$$

which at each slot specifies what are the system parameters to be used for that slot. It takes three parameters

$$\mathsf{AdvanceParameters}(\mathsf{Tree}, \mathsf{Block}, \mathsf{slot}) \ .$$

It is require that Block has a slot number larger or equal to $\mathsf{slot} - \mathsf{AdvanceTime}$. Then

$$\mathsf{AdvanceParameters}(\mathsf{Tree}, \mathsf{Block}, \mathsf{slot}) = \mathsf{SlotParameters}(\mathsf{Tree}, \mathsf{Block}, \mathsf{slot}^{\mathrm{ADV}}) ,$$

where

$$\mathsf{slot}^{\mathrm{ADV}} = \mathsf{slot} - \mathsf{AdvanceTime} .$$

We write $\mathsf{AdvanceParameters}(\mathsf{Block}, \mathsf{slot})$ for brevity.

$\triangle$

Having advance parameters is sufficient for the tree layer. It allows you to check in advance whether you win a slot you can use $\mathsf{AdvanceParameters}(\mathsf{Block}, \mathsf{slot})$ as soon as you know a block within time $\mathsf{AdvanceTime}$ of slot. Notice that when someone checks your block, they will use $\mathsf{AdvanceParameters}(\mathsf{Block}', \mathsf{slot})$ for another block within time $\mathsf{AdvanceTime}$ of slot. If Block and Block' are on the same path, then the parameters will be the same. If they are not on the same path, but on different branches, then the parameters used to check might not be the same as those used by you. So if you want to ensure that the future checker will acknowledge that you won, you better use a final block when you compute $\mathsf{AdvanceParameters}(\mathsf{Block}, \mathsf{slot})$. Therefore Block should not be the leaf of the tree, where branching is going on, but preferable a bit back in the tree. This is another reason to have $\mathsf{AdvanceTime}$ be large. We want to use a block Block from a bit back in time in $\mathsf{AdvanceParameters}(\mathsf{Block}, \mathsf{slot})$ when we predict the lottery and at the same time know the result well in advance.

17.1.4 Stable Parameters

There is a huge engineering disadvantage with $\mathsf{AdvanceParameters}$ if the parameters change often: each slot might have new parameters and a new slot might come every few milliseconds depending on how aggressive the system is tuned. We do not want to change parameters that often. This gives a new parameter $\mathsf{StableTime}$. It is how often we chance the parameters. Note that for positive integers t and d it holds that $t - (t \bmod d)$ is less than t and a multiple of d.

Definition 17.4 (stable parameters) We define a function

$$\mathsf{StableParameters}$$

which at each slot specifies what are the system parameters to be used for that slot. It takes three parameters

$$\mathsf{StableParameters}(\mathsf{Tree}, \mathsf{Block}, \mathsf{slot}) .$$

It is require that Block has a slot number larger or equal to $\mathsf{slot} - \mathsf{AdvanceTime}$. Then

$$\mathsf{StableParameters}(\mathsf{Tree}, \mathsf{Block}, \mathsf{slot}) = \mathsf{AdvanceParameters}(\mathsf{Tree}, \mathsf{Block}, \mathsf{slot}^{\mathrm{STBL}}) ,$$

where

$$\mathsf{slot}^{\mathrm{STBL}} = \mathsf{slot} - (\mathsf{slot} \bmod \mathsf{StableTime}) \ .$$

△

Exercise 17.1 (dynamic proof-of-stake) Start from your solution in Exercise 12.2 and add the following dynamic properties using StableParameters.

Dynamic stake As the AUs move around, make sure that the number of tickets used in te lottery follow the money.

Dynamic blocktime Dynamically adjust the blocktime such that a block is made every 10 second on average. Make Hardness a block parameter. If blocks are made too slowly, decrease the hardness. If blocks are being made to fast, increase the hardness.

Dynamic participation When more accounts get money on them, the system should handle that the peer with that accounts starts taking part in the lottery system. The main challenge is to ensure that new peers get up to date and receive all previous transactions and blocs.

17.1.5 Final Parameters

Using stable parameters is sufficient for the tree layer. But when you run the finalization layer, you *must* use final parameters. Recall that it is important that all parties run the finalization protocol at the same slots and with the same parameters. Therefore the votes to be used for a finalization run at slot slot must be read from a final block that all honest parties agree is final. For this we define FinalStableParameters. It is defined like StableParameters, but instead of defining slot parameters by going back to the last known block, we go back to the last known *final* block.

Definition 17.5 (final parameters) We define a function

$$\mathsf{FinalStableParameters}$$

which at each slot specifies what are the system parameters to be used for that slot. It takes three parameters

$$\mathsf{FinalStableParameters}(\mathsf{Tree}, \mathsf{Block}, \mathsf{slot}) \ .$$

We drop Tree for brevity. It is require that Block has a slot number larger or equal to $\mathsf{slot} - \mathsf{AdvanceTime}$. Then

$$\begin{aligned}
\mathsf{FinalBlockParameters}(\mathsf{Block}) &= \mathsf{BlockParameters}(\mathrm{PathTo}(\mathsf{Block})[\textsc{Block}.\mathsf{LastAgreed}]) \\
\mathsf{FinalSlotParameters}(\mathsf{Block}, \mathsf{slot}) &= \mathsf{FinalBlockParameters}(\mathsf{Block}_{\leq \mathsf{slot}}) \\
\mathsf{FinalAdvanceParameters}(\mathsf{Block}, \mathsf{slot}) &= \mathsf{FinalSlotParameters}(\mathsf{Block}, \mathsf{slot}^{\mathrm{ADV}}) \\
\text{where } \mathsf{slot}^{\mathrm{ADV}} &= \mathsf{slot} - \mathsf{AdvanceTime} \\
\mathsf{FinalStableParameters}(\mathsf{Block}, \mathsf{slot}) &= \mathsf{FinalAdvanceParameters}(\mathsf{Block}, \mathsf{slot}^{\mathrm{STBL}}) \\
\text{where } \mathsf{slot}^{\mathrm{STBL}} &= \mathsf{slot} - (\mathsf{slot} \bmod \mathsf{StableTime})
\end{aligned}$$

A bit more operationally, the above definitions says the following. If you want to know which parameters to use at slot, go back to the largest multiple of $\mathsf{StableTime}$ that is less than slot, call it $\mathsf{slot}^{\mathrm{STBL}}$. Then subtract $\mathsf{AdvanceTime}$ from that slot number to get $\mathsf{slot}^{\mathrm{ADV}}$. Then take the final block with the largest slot number which is less than or equal to $\mathsf{slot}^{\mathrm{ADV}}$. Read the parameters from that block.

17.2 Some Important Missing Details

There are several missing details before one has a full-fledged blockchain. We look at a few of them here.

17.2.1 New Accounts

When new accounts are created the account creator could pick the secret key sk_i such that $\mathsf{Draw}_{\mathsf{slot},i} = \mathsf{Sig}_{\mathsf{sk}_i}(\text{LOTTERY}, \mathsf{Seed}, \mathsf{slot})$ is higher on average on the coming slots. The account creator knows $\text{LOTTERY}, \mathsf{Seed}, \mathsf{slot}$ and hence can just try several keys sk_i until it find a very "lucky" one. To mitigate this attack, the seed Seed needs to be updated every now and then. An account must then in the lottery use a seed published *after* it was created. A new account therefore cannot take part in the lottery until a new seed was made. It is important that the corrupted parties cannot influence the seed creation too much. Otherwise they could just create seeds that are particularly "lucky" for their own secret keys. How to create such seeds securely is a longer story that we do not dive into here.

17.2.2 Consensus Layer Peeking

Recall that the parameters used by the tree layer and the finality layer are computed by the smart contract layer. For this purpose it is convenient to expose to the smart contract layer some value from the tree layer and the finality layer. This is done via peeking contracts. These are contracts in the smart contract layer which contains values from the consensus layer. A value V can be put in a peek contract if it is guaranteed that all honest parties agree on V. They simply add it as a transaction to the next block and reject blocks that report a wrong value of V.

An important peek is used to provide random values to the smart contract layer. Recall that whenever a party wins the lottery a unpredictable value

$$\mathsf{Sig}_{\mathsf{sk}_i}(\text{LOTTERY}, \mathsf{Seed}, \mathsf{slot})$$

is computed. If we want to generate randomness to be used by smart contracts, then the winner will also compute another signature

$$\mathsf{Sig}_{\mathsf{sk}_i}(\text{RANDOM}, \mathsf{Seed}, \mathsf{slot})$$

and include

$$\mathsf{Sig}_{\mathsf{sk}_i}(\text{RANDOM}, \mathsf{Seed}, \mathsf{slot})$$

in the block. The value

$$R_{\mathsf{slot}} = H(\mathsf{Sig}_{\mathsf{sk}_i}(\textsc{random}, \mathsf{Seed}, \mathsf{slot}))$$

is then computed and $(\mathsf{slot}, R_{\mathsf{slot}})$ placed in a peek contract. These random values can in turn be used to compute

$$\mathsf{BlockParameters}(\mathsf{Block}).\mathsf{Seed}$$

to provide fresh seeds for future lotteries. How to do this securely is out of scope of this book.

Consensus layer peeking can also be used to report values resulting from surveys run using Byzantine agreement. The value V is known by all honest parties and can therefore be placed in a peek contract. This could for instance the a survey of the current network delay time or some other network parameter.

17.2.3 Training your Blockchain

Having finalization also allows to play with the parameters of your blockchain on the fly. You could try to lower the Hardness to tune the blocktime (the time between blocks are generated), to make the blockchain run faster. If you go too low, the tree layer will start branching a lot, but the finalization layer will catch this. The Δ-value will grow. In response to this you could increase the blocktime again until the branching goes away. This will allow to run with the lowest possible blocktime at all times. In good network conditions it can be low, and when under attack it will become higher to protect the safety of the system. This can all be controlled from the smart contract layer using the block parameter contract and peek contracts.

18

Sharding (Work in Progress)

19

Secret Sharing: Replication with Privacy (Work in Progress)

20

Secure Multi-Party Computation (Work in Progress)

21

Replicated State Machines with Secret State (Work in Progress)

22

Mathematical Preliminaries (Work in Progress)

Contents

In this section we introduce basic mathematical notation as we use it in this book.

22.1 Logic

22.2 Sets

We use $\{e_1, e_2, \ldots, e_n\}$ to denote the set containing the elements $e_1, e_2, \ldots, e_n$. If A and B are sets, then we use

$$A \subseteq B$$

to denote that A is a subset of B. We use

$$A \nsubseteq B$$

to denote that A is *not* a subset of B. For instance

$$\{1, 2\} \subseteq \{1, 2, 3\}$$

and

$$\{1, 2, 3\} \subseteq \{1, 2, 3\}$$

and

$$\{1, 2, 3\} \nsubseteq \{1, 2\} \ .$$

Note that if

$$A \subseteq B$$

and

$$B \subseteq A$$

then

$$A = B \ .$$

22.3 Probability

If E is an event, then we use $\Pr[E]$ to denote the probability that E happens.

22.4 Induction

Back Material

List of Exercises

References

[1] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols (extended abstract). In Robert L. Probert, Nancy A. Lynch, and Nicola Santoro, editors, *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 17-19, 1983*, pages 27–30. ACM, 1983.

[2] Gabriel Bracha. An asynchronou [(n-1)/3]-resilient consensus protocol. In Tiko Kameda, Jayadev Misra, Joseph G. Peters, and Nicola Santoro, editors, *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing, Vancouver, B. C., Canada, August 27-29, 1984*, pages 154–162. ACM, 1984.

[3] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 136–145. IEEE Computer Society, 2001.

[4] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[5] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[6] Ueli Maurer. Constructive cryptography - A new paradigm for security definitions and proofs. In Sebastian Mödersheim and Catuscia Palamidessi, editors, *Theory of Security and Applications - Joint Workshop, TOSCA 2011, Saarbrücken, Germany, March 31 - April 1, 2011, Revised Selected Papers*, volume 6993 of *Lecture Notes in Computer Science*, pages 33–56. Springer, 2011.