ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed
Computing

Distributed Systems

Roger Wattenhofer

wattenhofer@ethz.ch

Autumn 2020

Chapter 1

Introduction to Distributed Systems

Why Distributed Systems?

Today's computing and information systems are inherently *distributed*. Many companies are operating on a global scale, with thousands or even millions of machines on all the continents. Data is stored in various data centers, computing tasks are performed on multiple machines. At the other end of the spectrum, also your mobile phone is a distributed system. Not only does it probably share some of your data with the cloud, the phone itself contains multiple processing and storage units. Your phone is a complicated distributed architecture.

Moreover, computers have come a long way. In the early 1970s, microchips featured a clock rate of roughly 1 MHz. Ten years later, in the early 1980s, you could get a computer with a clock rate of roughly 10 MHz. In the early 1990s, clock speed was around 100 MHz. In the early 2000s, the first 1 GHz processor was shipped to customers. In 2002 one could already buy a processor with a clock rate between 3 and 4 GHz. If you buy a new computer today, chances are that the clock rate is still between 3 and 4 GHz, since clock rates basically stopped increasing. Clock speed can apparently not go beyond a few GHz without running into physical issues such as overheating. Since 2003, computing architectures are mostly developing by the multi-core revolution. Computers are becoming more parallel, concurrent, and distributed.

Finally, data is more reliably stored on multiple geographically distributed machines. This way, the data can withstand regional disasters such as floods, fire, meteorites, or electromagnetic pulses, for instance triggered by solar superstorms. In addition, geographically distributed data is also safer from human attacks. Recently we learned that computer hardware is pretty insecure. Scary attacks exist, with scary names such as spectre, meltdown, rowhammer, memory deduplication. There are even attacks on hardware that is considered secure! If we store our data on multiple machines, it may be safe assuming hackers cannot attack all machines concurrently. Moreover, data and software replication also help availability, as computer systems do not need to be shut down for maintenance.

In summary, today almost all computer systems are distributed, for different reasons:

- Geography: Large organizations and companies are inherently geographically distributed, and a computer system needs to deal with this issue anyway.
- Parallelism: To speed up computation, we employ multicore processors or computing clusters.
- Reliability: Data is replicated on different machines to prevent data loss.
- Availability: Data is replicated on different machines to allow for access at any time, without bottlenecks, minimizing latency.

Even though distributed systems have many benefits, such as increased storage or computational power, they also introduce challenging *coordination* problems. Some say that going from one computer to two is a bit like having a second child. When you have one child and all cookies are gone from the cookie jar, you know who did it!

Coordination problems are so prevalent, they come with various flavors and names. Probably there is a term for every letter of the alphabet: agreement, blockchain, consensus, consistency, distributed ledger, event sourcing, fault-tolerance, etc.

Coordination problems will happen quite often in a distributed system. Even though every single *node* (node is a general term for anything that computes, e.g. a computer, a multiprocessor core, a network switch, etc.) of a distributed system will only fail once every few years, with millions of nodes, you can expect a failure every minute. On the bright side, one may hope that a distributed system may have enough redundancy to tolerate node failures and continue to work correctly.

Distributed Systems Overview

We introduce some basic techniques to building distributed systems, with a focus on fault-tolerance. We will study different protocols and algorithms that allow for fault-tolerant operation, and we will discuss practical systems that implement these techniques.

We will see different models (and even more combinations of models) that can be studied. We will not discuss them in detail now, but simply define them when we use them. Towards the end of the course a general picture should emerge, hopefully!

The focus is on protocols and systems that matter in practice. In other words, in this course, we do not discuss concepts because they are fun, but because they are practically relevant.

Nevertheless, have fun!

Chapter Notes

Many good textbooks have been written on the subject, e.g. [AW04, CGR11, CDKB11, Lyn96, Mul93, Ray13, TS01]. James Aspnes has written an excellent freely available script on distributed systems [Asp14]. Similarly to our course, these texts focus on large-scale distributed systems, and hence there is some overlap with our course. There are also some excellent textbooks focusing on small-scale multicore systems, e.g. [HS08].

Some chapters of this course have been developed in collaboration with (former) PhD students, see chapter notes for details. Many colleagues and students have helped to improve exercises and script. Thanks go to Georg Bachmeier, Pascal Bissig, Philipp Brandes, Christian Decker, Manuel Eichelberger, Klaus-Tycho Förster, Arthur Gervais, Pankaj Khanchandani, Barbara Keller, Rik Melis, Darya Melnyk, Tejaswi Nadahalli, Peter Robinson, Jakub Sliwinski, Selma Steinhoff, Julian Steger, David Stolz, and Saravanan Vijayakumaran. Jinchuan Chen, Qiang Lin, Yunzhi Xue, and Qing Zhu translated this text into Simplified Chinese, and along the way found improvements to the English version as well. Thanks!

Bibliography

[Asp14] James Aspnes. Notes on Theory of Distributed Systems, 2014.

[AW04] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition).* John Wiley Interscience, March 2004.

[CDKB11] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Distributed Systems: Concepts and Design.* Addison-Wesley Publishing Company, USA, 5th edition, 2011.

[CGR11] Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. *Introduction to Reliable and Secure Distributed Programming.* Springer Publishing Company, Incorporated, 2nd edition, 2011.

[HS08] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[Mul93] Sape Mullender, editor. *Distributed Systems (2nd Ed.).* ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[Ray13] Michel Raynal. *Distributed Algorithms for Message-Passing Systems.* Springer Publishing Company, Incorporated, 2013.

[TS01] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001.

Chapter 2

Fault-Tolerance & Paxos

How do you create a fault-tolerant distributed system? In this chapter we start out with simple questions, and, step by step, improve our solutions until we arrive at a system that works even under adverse circumstances, Paxos.

2.1 Client/Server

Definition 2.1 (node)**.** *We call a single actor in the system* ***node****. In a computer network the computers are the nodes, in the classical client-server model both the server and the client are nodes, and so on. If not stated otherwise, the total number of nodes in the system is n.*

Model 2.2 (message passing)**.** *In the* ***message passing model*** *we study distributed systems that consist of a set of nodes. Each node can perform local computations, and can send messages to every other node.*

Remarks:

- We start with two nodes, the smallest number of nodes in a distributed system. We have a *client* node that wants to "manipulate" data (e.g., store, update, . . .) on a remote *server* node.

Algorithm 2.3 Naïve Client-Server Algorithm

```
Client sends commands one at a time to server
```

Model 2.4 (message loss)**.** *In the message passing model with* ***message loss****, for* ***any*** *specific message, it is not guaranteed that it will arrive safely at the receiver.*

Remarks:

- A related problem is message corruption, i.e., a message is received but the content of the message is corrupted. In practice, in contrast to message loss, message corruption can be handled quite well, e.g. by including additional information in the message, such as a checksum.

- Algorithm 2.3 does not work correctly if there is message loss, so we need a little improvement.

Algorithm 2.5 Client-Server Algorithm with Acknowledgments

```
Client sends commands one at a time to server
Server acknowledges every command
If the client does not receive an acknowledgment within a reasonable time,
    the client resends the command
```

Remarks:

- Sending commands "one at a time" means that when the client sent command c, the client does not send any new command c' until it received an acknowledgment for c.
- Since not only messages sent by the client can be lost, but also acknowledgments, the client might resend a message that was already received and executed on the server. To prevent multiple executions of the same command, one can add a *sequence number* to each message, allowing the receiver to identify duplicates.
- This simple algorithm is the basis of many reliable protocols, e.g. TCP.
- The algorithm can easily be extended to work with multiple servers: The client sends each command to every server, and once the client received an acknowledgment from each server, the command is considered to be executed successfully.
- What about multiple clients?

Model 2.6 (variable message delay)**.** *In practice, messages might experience different transmission times, even if they are being sent between the same two nodes.*

Remarks:

- Throughout this chapter, we assume the variable message delay model.

Theorem 2.7. *If Algorithm 2.5 is used with multiple clients and multiple servers, the servers might see the commands in different order, leading to an inconsistent state.*

Proof. Assume we have two clients u_1 and u_2, and two servers s_1 and s_2. Both clients issue a command to update a variable x on the servers, initially $x = 0$. Client u_1 sends command $x = x + 1$ and client u_2 sends $x = 2 \cdot x$.

Let both clients send their message at the same time. With variable message delay, it can happen that s_1 receives the message from u_1 first, and s_2 receives the message from u_2 first.[1] Hence, s_1 computes $x = (0 + 1) \cdot 2 = 2$ and s_2 computes $x = (0 \cdot 2) + 1 = 1$.

□

[1] For example, u_1 and s_1 are (geographically) located close to each other, and so are u_2 and s_2.

Definition 2.8 (state replication). *A set of nodes achieves **state replication**, if all nodes execute a (potentially infinite) sequence of commands* $c_1, c_2, c_3, \ldots$*, in the same order.*

Remarks:

- State replication is a fundamental property for distributed systems.
- For people working in the financial tech industry, state replication is often synonymous with the term blockchain. The Bitcoin blockchain we will discuss in Chapter 9 is indeed one way to implement state replication. However, as we will see in all the other chapters, there are many alternative concepts that are worth knowing, with different properties.
- Since state replication is trivial with a single server, we can designate a single server as a *serializer*. By letting the serializer distribute the commands, we automatically order the requests and achieve state replication!

Algorithm 2.9 State Replication with a Serializer

```
Clients send commands one at a time to the serializer
Serializer forwards commands one at a time to all other servers
Once the serializer received all acknowledgments, it notifies the client about
the success
```

Remarks:

- This idea is sometimes also referred to as *leader/follower (or parent/child) replication.*
- What about node failures? Our serializer is a single point of failure!
- Can we have a more *distributed* approach of solving state replication? Instead of directly establishing a consistent order of commands, we can use a different approach: We make sure that there is always at most one client sending a command; i.e., we use *mutual exclusion*, respectively *locking.*

Algorithm 2.10 Two-Phase Protocol

```
Phase 1
Client asks all servers for the lock
Phase 2
if client receives lock from every server then
    Client sends command reliably to each server, and gives the lock back
else
    Clients gives the received locks back
    Client waits, and then starts with Phase 1 again
end if
```

Remarks:

- This idea appears in many contexts and with different names, usually with slight variations, e.g. *two-phase locking (2PL)*.
- Another example is the *two-phase commit (2PC)* protocol, typically presented in a database environment. The first phase is called the *preparation* of a transaction, and in the second phase the transaction is either *committed* or *aborted*. The 2PC process is not started at the client but at a designated server node that is called the *coordinator*.
- It is often claimed that 2PL and 2PC provide better consistency guarantees than a simple serializer if nodes can *recover* after crashing. In particular, alive nodes might be kept consistent with crashed nodes, for transactions that started while the crashed node was still running. This benefit was even improved in a protocol that uses an additional phase (3PC).
- The problem with 2PC or 3PC is that they are not well-defined if exceptions happen.
- Does Algorithm 2.10 really handle node crashes well? No! In fact, it is even worse than the simple serializer approach (Algorithm 2.9): Instead of needing one available node, Algorithm 2.10 requires *all* servers to be responsive!
- Does Algorithm 2.10 also work if we only get the lock from a subset of servers? Is a majority of servers enough?
- What if two or more clients concurrently try to acquire a majority of locks? Do clients have to abandon their already acquired locks, in order not to run into a deadlock? How? And what if they crash before they can release the locks?
- Bad news: It seems we need a slightly more complicated concept.
- Good news: We postpone the complexity of achieving state replication and first show how to execute a single command only.

2.2 Paxos

Definition 2.11 (ticket)**.** *A* ***ticket*** *is a weaker form of a lock, with the following properties:*

- **Reissuable:** *A server can issue a ticket, even if previously issued tickets have not yet been returned.*
- **Ticket expiration:** *If a client sends a message to a server using a previously acquired ticket t, the server will only accept t, if t is the most recently issued ticket.*

Remarks:

- There is no problem with crashes: If a client crashes while holding a ticket, the remaining clients are not affected, as servers can simply issue new tickets.
- Tickets can be implemented with a counter: Each time a ticket is requested, the counter is increased. When a client tries to use a ticket, the server can determine if the ticket is expired.
- What can we do with tickets? Can we simply replace the locks in Algorithm 2.10 with tickets? We need to add at least one additional phase, as only the client knows if a majority of the tickets have been valid in Phase 2.

Algorithm 2.12 Naïve Ticket Protocol

```
    Phase 1
 1: Client asks all servers for a ticket
    Phase 2
 2: if a majority of the servers replied then
 3:     Client sends command together with ticket to each server
 4:     Server stores command only if ticket is still valid, and replies to client
 5: else
 6:     Client waits, and then starts with Phase 1 again
 7: end if
    Phase 3
 8: if client hears a positive answer from a majority of the servers then
 9:     Client tells servers to execute the stored command
10: else
11:     Client waits, and then starts with Phase 1 again
12: end if
```

Remarks:

- There are problems with this algorithm: Let u_1 be the first client that successfully stores its command c_1 on a majority of the servers. Assume that u_1 becomes very slow just before it can notify the servers (Line 9), and a client u_2 updates the stored command in some servers to c_2. Afterwards, u_1 tells the servers to execute the command. Now some servers will execute c_1 and others c_2!
- How can this problem be fixed? We know that every client u_2 that updates the stored command after u_1 must have used a newer ticket than u_1. As u_1's ticket was accepted in Phase 2, it follows that u_2 must have acquired its ticket after u_1 already stored its value in the respective server.

- Idea: What if a server, instead of only handing out tickets in Phase 1, also notifies clients about its currently stored command? Then, u_2 learns that u_1 already stored c_1 and instead of trying to store c_2, u_2 could support u_1 by also storing c_1. As both clients try to store and execute the same command, the order in which they proceed is no longer a problem.

- But what if not all servers have the same command stored, and u_2 learns multiple stored commands in Phase 1. What command should u_2 support?

- Observe that it is always safe to support the most recently stored command. As long as there is no majority, clients can support any command. However, once there is a majority, clients need to support this value.

- So, in order to determine which command was stored most recently, servers can remember the ticket number that was used to store the command, and afterwards tell this number to clients in Phase 1.

- If every server uses its own ticket numbers, the newest ticket does not necessarily have the largest number. This problem can be solved if clients suggest the ticket numbers themselves!

Algorithm 2.13 Paxos

```
Client (Proposer)                              Server (Acceptor)

Initialization
$c$        ◁ command to execute                $T_{\text{max}} = 0$  ◁ largest issued ticket
$t = 0$    ◁ ticket number to try
                                               $C = \bot$            ◁ stored command
                                               $T_{\text{store}} = 0$ ◁ ticket used to store $C$

Phase 1
$t = t + 1$
Ask all servers for ticket $t$
                                               if $t > T_{\text{max}}$ then
                                                   $T_{\text{max}} = t$
                                                   Answer with ok($T_{\text{store}}$, $C$)
                                               end if

Phase 2
if a majority answers ok then
    Pick ($T_{\text{store}}$, $C$) with largest $T_{\text{store}}$

    if $T_{\text{store}} > 0$ then
        $c = C$
    end if
    Send propose($t$, $c$) to same
    majority
end if

                                               if $t = T_{\text{max}}$ then
                                                   $C = c$
                                                   $T_{\text{store}} = t$
                                                   Answer success
                                               end if
Phase 3

if a majority answers success
then
    Send execute($c$) to every server
end if
```

Remarks:

- Unlike previously mentioned algorithms, there is no step where a client explicitly decides to start a new attempt and jumps back to Phase 1. Note that this is not necessary, as a client can decide to abort the current attempt and start a new one *at any point* in the algorithm. This has the advantage that we do not need to be careful about selecting “good” values for timeouts, as correctness is independent of the decisions when to start new attempts.

- The performance can be improved by letting the servers send negative replies in phases 1 and 2 if the ticket expired.

- The contention between different clients can be alleviated by randomizing the waiting times between consecutive attempts.

Lemma 2.14. *We call a message* propose(t,c) *sent by clients on Line 12 a* ***proposal for (t,c)****. A proposal for (t,c) is* ***chosen****, if it is stored by a majority of servers (Line 15). For every issued* propose(t',c') *with* $t' > t$ *holds that* $c' = c$*, if there was a chosen* propose(t,c)*.*

Proof. Observe that there can be at most one proposal for every ticket number τ since clients only send a proposal if they received a majority of the tickets for τ (Line 7). Hence, every proposal is uniquely identified by its ticket number τ.

Assume that there is at least one propose(t',c') with $t' > t$ and $c' \neq c$; of such proposals, consider the proposal with the smallest ticket number t'. Since both this proposal and also the propose(t,c) have been sent to a majority of the servers, we can denote by S the non-empty intersection of servers that have been involved in both proposals. Since propose(t,c) has been chosen, this means that at least one server $s \in S$ must have stored command c; thus, when the command was stored, the ticket number t was still valid. Hence, s must have received the request for ticket t' *after* it already stored propose(t,c), as the request for ticket t' invalidates ticket t.

Therefore, the client that sent propose(t',c') must have learned from s that a client already stored propose(t,c). Since a client adapts its proposal to the command that is stored with the highest ticket number so far (Line 8), the client must have proposed c as well. There is only one possibility that would lead to the client not adapting c: If the client received the information from a server that some client stored propose(t^*,c^*), with $c^* \neq c$ and $t^* > t$. In this case, a client must have sent propose(t^*,c^*) with $t < t^* < t'$, but this contradicts the assumption that t' is the smallest ticket number of a proposal issued after t. □

Theorem 2.15. *If a command c is executed by some servers, all servers (eventually) execute c.*

Proof. From Lemma 2.14 we know that once a proposal for c is chosen, every subsequent proposal is for c. As there is exactly one first propose(t,c) that is chosen, it follows that all successful proposals will be for the command c. Thus, only proposals for a single command c can be chosen, and since clients only tell servers to execute a command, when it is chosen (Line 20), each client will eventually tell every server to execute c. □

Remarks:

- If the client with the first successful proposal does not crash, it will directly tell every server to execute c.

- However, if the client crashes before notifying any of the servers, the servers will execute the command only once the next client is successful. Once a server received a request to execute c, it can inform every client that arrives later that there is already a chosen command, so that the client does not waste time with the proposal process.

- Note that Paxos cannot make progress if half (or more) of the servers crash, as clients cannot achieve a majority anymore.
- The original description of Paxos uses three roles: Proposers, acceptors and learners. Learners have a trivial role: They do nothing, they just learn from other nodes which command was chosen.
- We assigned every node only one role. In some scenarios, it might be useful to allow a node to have multiple roles. For example in a peer-to-peer scenario nodes need to act as both client and server.
- Clients (Proposers) must be trusted to follow the protocol strictly. However, this is in many scenarios not a reasonable assumption. In such scenarios, the role of the proposer can be executed by a set of servers, and clients need to contact proposers, to propose values in their name.
- So far, we only discussed how a set of nodes can reach decision for a single command with the help of Paxos. We call such a single decision an *instance* of Paxos.
- For state replication as in Definition 2.8, we need to be able to execute multiple commands, we can extend each instance with an instance number, that is sent around with every message. Once the 1$^{\text{st}}$ command is chosen, any client can decide to start a new instance and compete for the 2$^{\text{nd}}$ command. If a server did not realize that the 1$^{\text{st}}$ instance already came to a decision, the server can ask other servers about the decisions to catch up.

Chapter Notes

Two-phase protocols have been around for a long time, and it is unclear if there is a single source of this idea. One of the earlier descriptions of this concept can found in the book of Gray [Gra78].

Leslie Lamport introduced Paxos in 1989. But why is it called Paxos? Lamport described the algorithm as the solution to a problem of the parliament of a fictitious Greek society on the island Paxos. He even liked this idea so much, that he gave some lectures in the persona of an Indiana-Jones-style archaeologist! When the paper was submitted, many readers were so distracted by the descriptions of the activities of the legislators, they did not understand the meaning and purpose of the algorithm. The paper was rejected. But Lamport refused to rewrite the paper, and he later wrote that he *"was quite annoyed at how humorless everyone working in the field seemed to be"*. A few years later, when the need for a protocol like Paxos arose again, Lamport simply took the paper out of the drawer and gave it to his colleagues. They liked it. So Lamport decided to submit the paper (in basically unaltered form!) again, 8 years after he wrote it – and it got accepted! But as this paper [Lam98] is admittedly hard to read, he had mercy, and later wrote a simpler description of Paxos [Lam01].

Leslie Lamport is an eminent scholar when it comes to understanding distributed systems, and we will learn some of his contributions in almost every chapter. Not surprisingly, Lamport has won the 2013 Turing Award for his fundamental contributions to the "theory and practice of distributed and concurrent systems, notably the invention of concepts such as causality and logical clocks, safety and liveness, replicated state machines, and sequential consistency" [Mal13]. One can add arbitrarily to this official citation, for instance Lamports popular LaTeX typesetting system, based on Donald Knuths TeX.

This chapter was written in collaboration with David Stolz.

Bibliography

[Gra78] James N Gray. *Notes on data base operating systems.* Springer, 1978.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[Lam01] Leslie Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[Mal13] Dahlia Malkhi. Leslie Lamport. ACM webpage, 2013.

Chapter 3

Consensus

3.1 Two Friends

Alice wants to arrange dinner with Bob, and since both of them are very reluctant to use the “call” functionality of their phones, she sends a text message suggesting to meet for dinner at 6pm. However, texting is unreliable, and Alice cannot be sure that the message arrives at Bob’s phone, hence she will only go to the meeting point if she receives a confirmation message from Bob. But Bob cannot be sure that his confirmation message is received; if the confirmation is lost, Alice cannot determine if Bob did not even receive her suggestion, or if Bob’s confirmation was lost. Therefore, Bob demands a confirmation message from Alice, to be sure that she will be there. But as this message can also be lost. . .

You can see that such a message exchange continues forever, if both Alice and Bob want to be sure that the other person will come to the meeting point!

Remarks:

- Such a protocol cannot terminate: Assume that there are protocols which lead to agreement, and P is one of the protocols which require the least number of messages. As the last confirmation might be lost and the protocol still needs to guarantee agreement, we can simply decide to always omit the last message. This gives us a new protocol P' which requires less messages than P, contradicting the assumption that P required the minimal amount of messages.
- Can Alice and Bob use Paxos?

3.2 Consensus

In Chapter 2 we studied a problem that we vaguely called agreement. We will now introduce a formally specified variant of this problem, called *consensus*.

Definition 3.1 (consensus)**.** *There are n nodes, of which at most f might crash, i.e., at least $n - f$ nodes are **correct**. Node i starts with an input value v_i. The nodes must decide for one of those values, satisfying the following properties:*

- **Agreement** *All correct nodes decide for the same value.*
- **Termination** *All correct nodes terminate in finite time.*
- **Validity** *The decision value must be the input value of a* node.

Remarks:

- We assume that every node can send messages to every other node, and that we have reliable links, i.e., a message that is sent will be received.
- There is no broadcast medium. If a node wants to send a message to multiple nodes, it needs to send multiple individual messages. If a node crashes while broadcasting, not all nodes may receive the broadcasted message. Later we will call this best-effort broadcast.
- Does Paxos satisfy all three criteria? If you study Paxos carefully, you will notice that Paxos does not guarantee termination. For example, the system can be stuck forever if two clients continuously request tickets, and neither of them ever manages to acquire a majority.
- One may hope to fix Paxos somehow, to guarantee termination. However, this is impossible. In fact, the consensus problem of Definition 3.1 cannot be solved by any algorithm.

3.3 Impossibility of Consensus

Model 3.2 (asynchronous)**.** *In the **asynchronous model**, algorithms are event based ("upon receiving message . . . , do . . . "). Nodes do not have access to a synchronized wall-clock. A message sent from one node to another will arrive in a finite but unbounded time.*

Remarks:

- The asynchronous time model is a widely used formalization of the variable message delay model (Model 2.6).

Definition 3.3 (asynchronous runtime)**.** *For algorithms in the asynchronous model, the **runtime** is the number of time units from the start of the execution to its completion in the worst case (every legal input, every execution scenario), assuming that each message has a delay of **at most** one time unit.*

Remarks:

- The maximum delay cannot be used in the algorithm design, i.e., the algorithm must work independent of the actual delay.
- Asynchronous algorithms can be thought of as systems, where local computation is significantly faster than message delays, and thus can be done in no time. Nodes are only active once an event occurs (a message arrives), and then they perform their actions "immediately".

- We will show now that crash failures in the asynchronous model can be quite harsh. In particular there is no deterministic fault-tolerant consensus algorithm in the asynchronous model, not even for binary input.

Definition 3.4 (configuration). *We say that a system is fully defined (at any point during the execution) by its* ***configuration*** C*. The configuration includes the state of every node, and all messages that are in transit (sent but not yet received).*

Definition 3.5 (univalent). *We call a configuration* C ***univalent****, if the decision value is determined independently of what happens afterwards.*

Remarks:

- We call a configuration that is univalent for value v *v-valent.*
- Note that a configuration can be univalent, even though no single node is aware of this. For example, the configuration in which all nodes start with value 0 is 0-valent (due to the validity requirement).
- As we restricted the input values to be binary, the decision value of any consensus algorithm will also be binary (due to the validity requirement).

Definition 3.6 (bivalent). *A configuration* C *is called* ***bivalent*** *if the nodes might decide for* 0 *or* 1*.*

Remarks:

- The decision value depends on the order in which messages are received or on crash events. I.e., the decision is not yet made.
- We call the initial configuration of an algorithm C_0. When nodes are in C_0, all of them executed their initialization code and possibly, based on their input values, sent some messages. These initial messages are also included in C_0. In other words, in C_0 the nodes are now waiting for the first message to arrive.

Lemma 3.7. *There is at least one selection of input values* V *such that the according initial configuration* C_0 *is bivalent, if* $f \geq 1$*.*

Proof. As explained in the previous remark, C_0 only depends on the input values of the nodes. Let $V = [v_0, v_1, \dots, v_{n-1}]$ denote the array of input values, where v_i is the input value of node i.

We construct $n+1$ arrays $V_0, V_1, \dots, V_n$, where the index i in V_i denotes the position in the array up to which all input values are 1. So, $V_0 = [0, 0, 0, \dots, 0]$, $V_1 = [1, 0, 0, \dots, 0]$, and so on, up to $V_n = [1, 1, 1, \dots, 1]$.

Note that the configuration corresponding to V_0 must be 0-valent so that the validity requirement is satisfied. Analogously, the configuration corresponding to V_n must be 1-valent. Assume that all initial configurations with starting values V_i are univalent. Therefore, there must be at least one index b, such that the configuration corresponding to V_{b-1} is 0-valent, and configuration corresponding to V_b is 1-valent. Observe that only the input value of the b^{th} node differs from V_{b-1} to V_b.

Since we assumed that the algorithm can tolerate at least one failure, i.e., $f \geq 1$, we look at the following execution: All nodes except b start with their initial value according to V_{b-1} respectively V_b. Node b is "extremely slow"; i.e., all messages sent by b are scheduled in such a way, that all other nodes must assume that b crashed, in order to satisfy the termination requirement. Since the nodes cannot determine the value of b, and we assumed that all initial configurations are univalent, they will decide for a value v independent of the initial value of b. Since V_{b-1} is 0-valent, v must be 0. However we know that V_b is 1-valent, thus v must be 1. Since v cannot be both 0 and 1, we have a contradiction.

□

Definition 3.8 (transition)**.** *A **transition** from configuration C to a following configuration C_τ is characterized by an event $\tau = (u, m)$, i.e., node u receiving message m.*

Remarks:

- Transitions are the formally defined version of the "events" in the asynchronous model we described before.
- A transition $\tau = (u, m)$ is only applicable to C, if m was still in transit in C.
- C_τ differs from C as follows: m is no longer in transit, u has possibly a different state (as u can update its state based on m), and there are (potentially) new messages in transit, sent by u.

Definition 3.9 (configuration tree)**.** *The **configuration tree** is a directed tree of configurations. Its root is the configuration C_0 which is fully characterized by the input values V. The edges of the tree are the transitions; every configuration has all applicable transitions as outgoing edges.*

Remarks:

- For any algorithm, there is exactly *one* configuration tree for every selection of input values.
- Leaves are configurations where the execution of the algorithm terminated. Note that we use termination in the sense that the system as a whole terminated, i.e., there will not be any transition anymore.
- Every path from the root to a leaf is one possible asynchronous execution of the algorithm.
- Leaves must be univalent, or the algorithm terminates without agreement.
- If a node u crashes when the system is in C, all transitions $(u, *)$ are removed from C in the configuration tree.

Lemma 3.10. *Assume two transitions $\tau_1 = (u_1, m_1)$ and $\tau_2 = (u_2, m_2)$ for $u_1 \neq u_2$ are both applicable to C. Let $C_{\tau_1\tau_2}$ be the configuration that follows C by first applying transition τ_1 and then τ_2, and let $C_{\tau_2\tau_1}$ be defined analogously. It holds that $C_{\tau_1\tau_2} = C_{\tau_2\tau_1}$.*

Proof. Observe that τ_2 is applicable to C_{τ_1}, since m_2 is still in transit and τ_1 cannot change the state of u_2. With the same argument τ_1 is applicable to C_{τ_2}, and therefore both $C_{\tau_1\tau_2}$ and $C_{\tau_2\tau_1}$ are well-defined. Since the two transitions are completely independent of each other, meaning that they consume the same messages, lead to the same state transitions and to the same messages being sent, it follows that $C_{\tau_1\tau_2} = C_{\tau_2\tau_1}$. □

Definition 3.11 (critical configuration)**.** *We say that a configuration C is* ***critical****, if C is bivalent, but all configurations that are direct children of C in the configuration tree are univalent.*

Remarks:

- Informally, C is critical, if it is the last moment in the execution where the decision is not yet clear. As soon as the next message is processed by any node, the decision will be determined.

Lemma 3.12. *If a system is in a bivalent configuration, it must reach a critical configuration within finite time, or it does not always solve consensus.*

Proof. Recall that there is at least one bivalent initial configuration (Lemma 3.7). Assuming that this configuration is not critical, there must be at least one bivalent following configuration; hence, the system may enter this configuration. But if this configuration is not critical as well, the system may afterwards progress into another bivalent configuration. As long as there is no critical configuration, an unfortunate scheduling (selection of transitions) can always lead the system into another bivalent configuration. The only way how an algorithm can *enforce* to arrive in a univalent configuration is by reaching a critical configuration.

Therefore we can conclude that a system which does not reach a critical configuration has at least one possible execution where it will terminate in a bivalent configuration (hence it terminates without agreement), or it will not terminate at all.

□

Lemma 3.13. *If a configuration tree contains a critical configuration, crashing a single node can create a bivalent leaf; i.e., a crash prevents the algorithm from reaching agreement.*

Proof. Let C denote critical configuration in a configuration tree, and let T be the set of transitions applicable to C. Let $\tau_0 = (u_0, m_0) \in T$ and $\tau_1 = (u_1, m_1) \in T$ be two transitions, and let C_{τ_0} be 0-valent and C_{τ_1} be 1-valent. Note that T must contain these transitions, as C is a critical configuration.

Assume that $u_0 \neq u_1$. Using Lemma 3.10 we know that C has a following configuration $C_{\tau_0\tau_1} = C_{\tau_1\tau_0}$. Since this configuration follows C_{τ_0} it must be 0-valent. However, this configuration also follows C_{τ_1} and must hence be 1-valent. This is a contradiction and therefore $u_0 = u_1$ must hold.

Therefore we can pick one particular node u for which there is a transition $\tau = (u, m) \in T$ which leads to a 0-valent configuration. As shown before, all transitions in T which lead to a 1-valent configuration must also take place on u. Since C is critical, there must be at least one such transition. Applying the same argument again, it follows that all transitions in T that lead to a 0-valent configuration must take place on u as well, and since C is critical, there is no transition in T that leads to a bivalent configuration. Therefore *all* transitions applicable to C take place on the *same* node u!

If this node u crashes while the system is in C, *all transitions are removed*, and therefore the system is stuck in C, i.e., it terminates in C. But as C is critical, and therefore bivalent, the algorithm fails to reach an agreement.

□

Theorem 3.14. *There is no deterministic algorithm which always achieves consensus in the asynchronous model, with $f > 0$.*

Proof. We assume that the input values are binary, as this is the easiest non-trivial possibility. From Lemma 3.7 we know that there must be at least one bivalent initial configuration C. Using Lemma 3.12 we know that if an algorithm solves consensus, all executions starting from the bivalent configuration C must reach a critical configuration. But if the algorithm reaches a critical configuration, a single crash can prevent agreement (Lemma 3.13). □

Remarks:

- If $f = 0$, then each node can simply send its value to all others, wait for all values, and choose the minimum.
- But if a single node may crash, there is no deterministic solution to consensus in the asynchronous model.
- How can the situation be improved? For example by giving each node access to randomness, i.e., we allow each node to toss a coin.

3.4 Randomized Consensus

Algorithm 3.15 Randomized Consensus (assuming $f < n/2$)

```
 1: $v_i \in \{0, 1\}$            ◁ input bit
 2: round = 1
 3: while true do
 4:     Broadcast myValue($v_i$, round)
        Propose
 5:     Wait until a majority of myValue messages of current round arrived
 6:     if all messages contain the same value $v$ then
 7:         Broadcast propose($v$, round)
 8:     else
 9:         Broadcast propose($\perp$, round)
10:     end if
        Vote
11:     Wait until a majority of propose messages of current round arrived
12:     if all messages propose the same value $v$ then
13:         Broadcast myValue($v$, round + 1)
14:         Broadcast propose($v$, round + 1)
15:         Decide for $v$ and terminate
16:     else if there is at least one proposal for $v$ then
17:         $v_i = v$
18:     else
19:         Choose $v_i$ randomly, with $Pr[v_i = 0] = Pr[v_i = 1] = 1/2$
20:     end if
21:     round = round + 1
22: end while
```

Remarks:

- The idea of Algorithm 3.15 is very simple: Either all nodes start with the same input bit, which makes consensus easy. Otherwise, nodes toss a coin until a large number of nodes get – by chance – the same outcome.

Lemma 3.16. *As long as no node decides and terminates, Algorithm 3.15 does not get stuck, independent of which nodes crash.*

Proof. The only two steps in the algorithm when a node waits are in Lines 5 and 11. Since a node only waits for a majority of the nodes to send a message, and since $f < n/2$, the node will always receive enough messages to continue, as long as no correct node terminates. □

Lemma 3.17. *Algorithm 3.15 satisfies the validity requirement.*

Proof. Observe that the validity requirement of consensus, when restricted to binary input values, corresponds to: If all nodes start with v, then v must be chosen; otherwise, either 0 or 1 is acceptable, and the validity requirement is automatically satisfied.

Assume that all nodes start with v. In this case, all nodes propose v in the first round. As all nodes only hear proposals for v, all nodes decide for v (Line 15) and terminate in the same round. □

Lemma 3.18. *Algorithm 3.15 satisfies the agreement requirement.*

Proof. Observe that proposals for both 0 and 1 cannot occur in the same round, as nodes only send a proposal for v, if they hear a *majority* for v in Line 5.

Let u be the first node that decides for a value v in round r. Hence, it received a majority of proposals for v in r (Line 7). Note that once a node receives a majority of proposals for a value, it will adapt this value and terminate in the same round. Since there cannot be a proposal for any other value in r, it follows that no node decides for a different value in r.

In Lemma 3.16 we only showed that nodes do not get stuck as long as no node decides, thus we need to be careful that no node gets stuck if u terminates.

Any node $u' \neq u$ can experience one of two scenarios: Either it also receives a majority for v in round r and terminates, or it does not receive a majority. In the first case, the agreement requirement is directly satisfied, and also the node cannot get stuck. Let us study the latter case. Since u heard a majority of proposals for v, it follows that every node hears *at least one* proposal for v. Hence, all nodes set their value v_i to v in round r. The nodes that terminate in round r also send one additional `myValue` and one `propose` message (Lines 13, 14). Therefore, all nodes will broadcast v at the beginning of round $r+1$, all nodes will propose v in the same round and, finally, all nodes will decide for the same value v. □

Lemma 3.19. *Algorithm 3.15 satisfies the termination requirement, i.e., all nodes terminate in expected time $O(2^n)$.*

Proof. We know from the proof of Lemma 3.18 that once a node hears a majority of proposals for a value, all nodes will terminate at most one round later. Hence, we only need to show that a node receives a majority of proposals for the same value within expected time $O(2^n)$.

Assume that no node receives a majority of proposals for the same value. In such a round, some nodes may update their value to v based on a proposal (Line 17). As shown before, all nodes that update the value based on a proposal, adapt the same value v. The rest of the nodes chooses 0 or 1 randomly. The probability that all nodes choose the same value v in one round is hence at least $1/2^n$. Therefore, the expected number of rounds is bounded by $O(2^n)$. As every round consists of two message exchanges, the asymptotic runtime of the algorithm is equal to the number of rounds. □

Theorem 3.20. *Algorithm 3.15 achieves binary consensus with expected runtime $O(2^n)$ if up to $f < n/2$ nodes crash.*

Remarks:

- How good is a fault tolerance of $f < n/2$?

Theorem 3.21. *There is no consensus algorithm for the asynchronous model that tolerates $f \geq n/2$ many failures.*

Proof. Assume that there is an algorithm that can handle $f = n/2$ many failures. We partition the set of all nodes into two sets N, N' both containing $n/2$ many nodes. Let us look at three different selection of input values: In V_0 all nodes start with 0. In V_1 all nodes start with 1. In V_{half} all nodes in N start with 0, and all nodes in N' start with 1.

Assume that nodes start with V_{half}. Since the algorithm must solve consensus independent of the scheduling of the messages, we study the scenario where all messages sent from nodes in N to nodes in N' (or vice versa) are heavily delayed. Note that the nodes in N cannot determine if they started with V_0 or V_{half}. Analogously, the nodes in N' cannot determine if they started in V_1 or V_{half}. Hence, if the algorithm terminates before any message from the other set is received, N must decide for 0 and N' must decide for 1 (to satisfy the validity requirement, as they could have started with V_0 respectively V_1). Therefore, the algorithm would fail to reach agreement.

The only possibility to overcome this problem is to wait for at least one message sent from a node of the other set. However, as $f = n/2$ many nodes can crash, the entire other set could have crashed before they sent any message. In that case, the algorithm would wait forever and therefore not satisfy the termination requirement.

□

Remarks:

- Algorithm 3.15 solves consensus with optimal fault-tolerance – but it is awfully slow. The problem is rooted in the individual coin tossing: If all nodes toss the same coin, they could terminate in a constant number of rounds.

- Can this problem be fixed by simply always choosing 1 at Line 19?!

- This cannot work: Such a change makes the algorithm deterministic, and therefore it cannot achieve consensus (Theorem 3.14). Simulating what happens by always choosing 1, one can see that it might happen that there is a majority for 0, but a minority with value 1 prevents the nodes from reaching agreement.

- Nevertheless, the algorithm can be improved by tossing a so-called *shared coin*. A shared coin is a random variable that is 0 for all nodes with constant probability, and 1 with constant probability. Of course, such a coin is not a magic device, but it is simply an algorithm. To improve the expected runtime of Algorithm 3.15, we replace Line 19 with a function call to the shared coin algorithm.

3.5 Shared Coin

Algorithm 3.22 Shared Coin (code for node u)

```
1: Choose local coin $c_u = 0$ with probability $1/n$, else $c_u = 1$
2: Broadcast myCoin($c_u$)

3: Wait for $n - f$ coins and store them in the local coin set $C_u$
4: Broadcast mySet($C_u$)

5: Wait for $n - f$ coin sets
6: if at least one coin is 0 among all coins in the coin sets then
7:    return 0
8: else
9:    return 1
10: end if
```

Remarks:

- Since at most f nodes crash, all nodes will always receive $n - f$ coins respectively coin sets in Lines 3 and 5. Therefore, all nodes make progress and termination is guaranteed.
- We show the correctness of the algorithm for $f < n/3$. To simplify the proof we assume that $n = 3f + 1$, i.e., we assume the worst case.

Lemma 3.23. *Let u be a node, and let W be the set of coins that u received in at least $f + 1$ different coin sets. It holds that $|W| \geq f + 1$.*

Proof. Let C be the multiset of coins received by u. Observe that u receives exactly $|C| = (n-f)^2$ many coins, as u waits for $n-f$ coin sets each containing $n - f$ coins.

Assume that the lemma does not hold. Then, at most f coins are in all $n-f$ coin sets, and all other coins $(n - f)$ are in at most f coin sets. In other words, the total number of coins that u received is bounded by

$$|C| \leq f \cdot (n - f) + (n - f) \cdot f = 2f(n - f).$$

Our assumption was that $n > 3f$, i.e., $n-f > 2f$. Therefore $|C| \leq 2f(n-f) < (n - f)^2 = |C|$, which is a contradiction. □

Lemma 3.24. *All coins in W are seen by all correct nodes.*

Proof. Let $w \in W$ be such a coin. By definition of W we know that w is in at least $f + 1$ sets received by u. Since every other node also waits for $n - f$ sets before terminating, each node will receive at least one of these sets, and hence w must be seen by every node that terminates. □

Theorem 3.25. *If $f < n/3$ nodes crash, Algorithm 3.22 implements a shared coin.*

Proof. Let us first bound the probability that the algorithm returns 1 for all nodes. With probability $(1 - 1/n)^n \approx 1/e \approx 0.37$ all nodes chose their local coin equal to 1 (Line 1), and in that case 1 will be decided. This is only a lower bound on the probability that all nodes return 1, as there are also other scenarios based on message scheduling and crashes which lead to a global decision for 1. But a probability of 0.37 is good enough, so we do not need to consider these scenarios.

With probability $1 - (1 - 1/n)^{|W|}$ there is at least one 0 in W. Using Lemma 3.23 we know that $|W| \geq f + 1 \approx n/3$, hence the probability is about $1 - (1 - 1/n)^{n/3} \approx 1 - (1/e)^{1/3} \approx 0.28$. We know that this 0 is seen by all nodes (Lemma 3.24), and hence everybody will decide 0. Thus Algorithm 3.22 implements a shared coin. □

Remarks:

- We only proved the worst case. By choosing f fairly small, it is clear that $f + 1 \not\approx n/3$. However, Lemma 3.23 can be proved for $|W| \geq n - 2f$. To prove this claim you need to substitute the expressions in the contradictory statement: At most $n - 2f - 1$ coins can be in all $n - f$ coin sets, and $n - (n - 2f - 1) = 2f + 1$ coins can be in at most f coin sets. The remainder of the proof is analogous, the only difference is that the math is not as neat. Using the modified Lemma we know that $|W| \geq n/3$, and therefore Theorem 3.25 also holds for *any* $f < n/3$.
- We implicitly assumed that message scheduling was random; if we need a 0 but the nodes that want to propose 0 are "slow", nobody is going to see these 0's, and we do not have progress. There exist more complicated protocols that solve this problem.

Theorem 3.26. *Plugging Algorithm 3.22 into Algorithm 3.15 we get a randomized consensus algorithm which terminates in a constant expected number of rounds tolerating up to $f < n/3$ crash failures.*

Chapter Notes

The problem of two friends arranging a meeting was presented and studied under many different names; nowadays, it is usually referred to as the *Two Generals Problem*. The impossibility proof was established in 1975 by Akkoyunlu et al. [AEH75].

The proof that there is no deterministic algorithm that always solves consensus is based on the proof of Fischer, Lynch and Paterson [FLP85], known as FLP, which they established in 1985. This result was awarded the 2001 PODC Influential Paper Award (now called Dijkstra Prize). The idea for the randomized consensus algorithm was originally presented by Ben-Or [Ben83]. The concept of a shared coin was introduced by Bracha [Bra87]. The shared coin algorithm in this chapter was proposed by [AW04]and it assumes randomized scheduling. A shared coin that can withstand worst-case scheduling has been developed by Alistarh et al. [AAKS14]; this shared coin was inspired by earlier shared coin solutions in the shared memory model [Cha96].

Apart from randomization, there are other techniques to still get consensus. One possibility is to drop asynchrony and rely on time more, e.g. by assuming partial synchrony [DLS88] or timed asynchrony [CF98]. Another possibility is to add failure detectors [CT96].

This chapter was written in collaboration with David Stolz.

Bibliography

[AAKS14] Dan Alistarh, James Aspnes, Valerie King, and Jared Saia. Communication-efficient randomized consensus. In *28th International Symposium of Distributed Computing (DISC), Austin, TX, USA, October 12-15, 2014*, pages 61–75, 2014.

[AEH75] EA Akkoyunlu, K Ekanadham, and RV Huber. Some constraints and tradeoffs in the design of network communications. In *ACM SIGOPS Operating Systems Review*, volume 9, pages 67–74. ACM, 1975.

[AW04] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics (2nd edition).* John Wiley Interscience, March 2004.

[Ben83] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30. ACM, 1983.

[Bra87] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

[CF98] Flaviu Cristian and Christof Fetzer. The timed asynchronous distributed system model. In *Digest of Papers: FTCS-28, The Twenty-Eigth Annual International Symposium on Fault-Tolerant Computing, Munich, Germany, June 23-25, 1998*, pages 140–149, 1998.

[Cha96] Tushar Deepak Chandra. Polylog randomized wait-free consensus. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA*, pages 166–175, 1996.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[DLS88] Cynthia Dwork, Nancy A. Lynch, and Larry J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, 1985.

Chapter 4

Byzantine Agreement

In order to make flying safer, researchers studied possible failures of various sensors and machines used in airplanes. While trying to model the failures, they were confronted with the following problem: Failing machines did not just crash, instead they sometimes showed an unusual behavior before stopping completely. With these insights researchers proposed a more general failure model.

Definition 4.1 (Byzantine)**.** *A node which can have arbitrary behavior is called **byzantine**. This includes "anything imaginable", e.g., not sending any messages at all, or sending different and wrong messages to different neighbors, or lying about the input value.*

Remarks:

- Byzantine behavior also includes collusion, i.e., all byzantine nodes are being controlled by the same adversary.
- We assume that any two nodes communicate directly, and that no node can forge an incorrect sender address. This is a requirement, such that a single byzantine node cannot simply impersonate all nodes!
- We call non-byzantine nodes *correct* nodes.

Definition 4.2 (Byzantine Agreement)**.** *Finding consensus as in Definition 3.1 in a system with byzantine nodes is called **byzantine agreement**. An algorithm is f-resilient if it still works correctly with f byzantine nodes.*

Remarks:

- As for consensus (Definition 3.1) we also need agreement, termination and validity. Agreement and termination are straight-forward, but what about validity?

4.1 Validity

Definition 4.3 (Any-Input Validity)**.** *The decision value must be the input value of* ***any*** *node.*

Remarks:

- This is the validity definition we used for consensus, in Definition 3.1.
- Does this definition still make sense in the presence of byzantine nodes? What if byzantine nodes lie about their inputs?
- We would wish for a validity definition that differentiates between byzantine and correct inputs.

Definition 4.4 (Correct-Input Validity)**.** *The decision value must be the input value of a* ***correct*** *node.*

Remarks:

- Unfortunately, implementing correct-input validity does not seem to be easy, as a byzantine node following the protocol but lying about its input value is indistinguishable from a correct node. Here is an alternative.

Definition 4.5 (All-Same Validity)**.** *If* ***all*** *correct nodes start with the same input v, the decision value must be v.*

Remarks:

- If the decision values are binary, then correct-input validity is induced by all-same validity.
- If the input values are not binary, but for example from sensors that deliver values in $\mathbb{R}$, all-same validity is in most scenarios not really useful.

Definition 4.6 (Median Validity)**.** *If the input values are orderable, e.g. $v \in \mathbb{R}$, byzantine outliers can be prevented by agreeing on a value close to the* ***median*** *of the correct input values – how close depends on the number of byzantine nodes f.*

Remarks:

- Is byzantine agreement possible? If yes, with what validity condition?
- Let us try to find an algorithm which tolerates 1 single byzantine node, first restricting to the so-called synchronous model.

Model 4.7 (synchronous)**.** *In the* ***synchronous model****, nodes operate in synchronous rounds. In each round, each node may send a message to the other nodes, receive the messages sent by the other nodes, and do some local computation.*

Definition 4.8 (synchronous runtime)**.** *For algorithms in the synchronous model, the* ***runtime*** *is simply the number of rounds from the start of the execution to its completion in the worst case (every legal input, every execution scenario).*

4.2 How Many Byzantine Nodes?

Algorithm 4.9 Byzantine Agreement with $f = 1$.

```
1: Code for node $u$, with input value $x$:
   Round 1
2: Send tuple($u, x$) to all other nodes
3: Receive tuple($v, y$) from all other nodes $v$
4: Store all received tuple($v, y$) in a set $S_u$
   Round 2
5: Send set $S_u$ to all other nodes
6: Receive sets $S_v$ from all nodes $v$
7: $T$ = set of tuple($v, y$) seen in at least two sets $S_v$, including own $S_u$
8: Let tuple($v, y$) $\in T$ be the tuple with the smallest value $y$
9: Decide on value $y$
```

Remarks:

- Byzantine nodes may not follow the protocol and send syntactically incorrect messages. Such messages can easily be detected and discarded. It is worse if byzantine nodes send syntactically correct messages, but with bogus content, e.g., they send different messages to different nodes.
- Some of these mistakes cannot easily be detected: For example, if a byzantine node sends different values to different nodes in the first round; such values will be put into S_u. However, some mistakes can and must be detected: Observe that all nodes only relay information in Round 2, and do not repeat their own value. So, if a byzantine node sends a set S_v which contains a tuple(v, y), this tuple must be removed by u from S_v upon receiving it (Line 6).
- Recall that we assumed that nodes cannot forge their source address; thus, if a node receives tuple(v, y) in Round 1, it is guaranteed that this message was sent by v.

Lemma 4.10. *If $n \geq 4$, all correct nodes have the same set T.*

Proof. With $f = 1$ and $n \geq 4$ we have at least 3 correct nodes. A correct node will see every correct value at least twice, once directly from another correct node, and once through the third correct node. So all correct values are in T. If the byzantine node sends the same value to at least 2 other (correct) nodes, all correct nodes will see the value twice, so all add it to set T. If the byzantine node sends all different values to the correct nodes, none of these values will end up in any set T. □

Theorem 4.11. *Algorithm 4.9 reaches byzantine agreement if $n \geq 4$.*

Proof. We need to show agreement, any-input validity and termination. With Lemma 4.10 we know that all correct nodes have the same set T, and therefore agree on the same minimum value. The nodes agree on a value proposed by any node, so any-input validity holds. Moreover, the algorithm terminates after two rounds. □

Remarks:

- If $n > 4$ the byzantine node can put multiple values into T.
- Algorithm 4.9 only provides any-input agreement, which is questionable in the byzantine context: Assume a byzantine node sends different values to different nodes, what is its input value in that case?
- Algorithm 4.9 can be slightly modified to achieve all-same validity by choosing the smallest value that occurs at least twice.
- The idea of this algorithm can be generalized for any f and $n > 3f$. In the generalization, every node sends in every of $f + 1$ rounds all information it learned so far to all other nodes. In other words, message size increases exponentially with f.
- Does Algorithm 4.9 also work with $n = 3$?

Theorem 4.12. *Three nodes cannot reach byzantine agreement with all-same validity if one node among them is byzantine.*

Proof. We will assume that the three nodes satisfy all-same validity and show that they will violate the agreement condition under this assumption.

In order to achieve all-same validity, nodes have to deterministically decide for a value x if it is the input value of every correct node. Recall that a Byzantine node which follows the protocol is indistinguishable from a correct node. Assume a correct node sees that $n-f$ nodes including itself have an input value x. Then, by all-same validity, this correct node must deterministically decide for x.

In the case of three nodes ($n - f = 2$), a node has to decide on its own input value if another node has the same input value. Let us call the three nodes u, v and w. If correct node u has input 0 and correct node v has input 1, the byzantine node w can fool them by telling u that its value is 0 and simultaneously telling v that its value is 1. By all-same validity, this leads to u and v deciding on two different values, which violates the agreement condition. Even if u talks to v, and they figure out that they have different assumptions about w's value, u cannot distinguish whether w or v is byzantine. □

Theorem 4.13. *A network with n nodes cannot reach byzantine agreement with $f \geq n/3$ byzantine nodes.*

Proof. Assume (for the sake of contradiction) that there exists an algorithm A that reaches byzantine agreement for n nodes with $f \geq \lceil n/3 \rceil$ byzantine nodes. We will show that A cannot satisfy all-same validity and agreement simultaneously.

Let us divide the n nodes into three groups of size $n/3$ (either $\lfloor n/3 \rfloor$ or $\lceil n/3 \rceil$, if n is not divisible by 3). Assume that one group of size $\lceil n/3 \rceil \geq n/3$ contains only Byzantine and the other two groups only correct nodes. Let one group of correct nodes start with input value 0 and the other with input value 1. As in Lemma 4.12, the group of Byzantine nodes supports the input value of each node, so each correct node observes at least $n - f$ nodes who support its own input value. Because of all-same validity, every correct node has to deterministically decide on its own input value. Since the two groups of correct nodes had different input values, the nodes will decide on different values respectively, thus violating the agreement property. □

4.3 The King Algorithm

Algorithm 4.14 King Algorithm (for $f < n/3$)

```
1:  $x$ = my input value
2:  for phase = 1 to $f + 1$ do
        Vote
3:      Broadcast value($x$)
        Propose
4:      if some value($y$) received at least $n - f$ times then
5:          Broadcast propose($y$)
6:      end if
7:      if some propose($z$) received more than $f$ times then
8:          $x = z$
9:      end if
        King
10:     Let node $v_i$ be the predefined king of this phase $i$
11:     The king $v_i$ broadcasts its current value $w$
12:     if received strictly less than $n - f$ propose($y$) then
13:         $x = w$
14:     end if
15: end for
```

Lemma 4.15. *Algorithm 4.14 fulfills the all-same validity.*

Proof. If all correct nodes start with the same value, all correct nodes propose it in Line 5. All correct nodes will receive at least $n - f$ proposals, i.e., all correct nodes will stick with this value, and never change it to the king's value. This holds for all phases. □

Lemma 4.16. *If a correct node proposes x, no other correct node proposes y, with $y \neq x$, if $n > 3f$.*

Proof. Assume (for the sake of contradiction) that a correct node proposes value x and another correct node proposes value y. Since a good node only proposes a value if it heard at least $n - f$ `value` messages, we know that both nodes must have received their value from at least $n - 2f$ distinct correct nodes (as at most f nodes can behave byzantine and send x to one node and y to the other one). Hence, there must be a total of at least $2(n - 2f) + f = 2n - 3f$ nodes in the system. Using $3f < n$, we have $2n - 3f > n$ nodes, a contradiction. □

Lemma 4.17. *There is at least one phase with a correct king.*

Proof. There are $f + 1$ phases, each with a different king. As there are only f byzantine nodes, one king must be correct. □

Lemma 4.18. *After a phase with a correct king, the correct nodes will not change their values v anymore, if $n > 3f$.*

Proof. If all correct nodes change their values to the king's value, all correct nodes have the same value. If some correct node does not change its value to the king's value, it received a proposal at least $n - f$ times, therefore at least $n-2f$ correct nodes broadcasted this proposal. Thus, all correct nodes received it at least $n - 2f > f$ times (using $n > 3f$), therefore all correct nodes set their value to the proposed value, including the correct king. Note that only one value can be proposed more than f times, which follows from Lemma 4.16. With Lemma 4.15, no node will change its value after this phase. □

Theorem 4.19. *Algorithm 4.14 solves byzantine agreement.*

Proof. The king algorithm reaches agreement as either all correct nodes start with the same value, or they agree on the same value latest after the phase where a correct node was king according to Lemmas 4.17 and 4.18. Because of Lemma 4.15 we know that they will stick with this value. Termination is guaranteed after $3(f + 1)$ rounds, and all-same validity is proved in Lemma 4.15. □

Remarks:

- Algorithm 4.14 requires $f + 1$ predefined kings. We assume that the kings (and their order) are given. Finding the kings indeed would be a byzantine agreement task by itself, so this must be done before the execution of the King algorithm.
- Do algorithms exist which do not need predefined kings? Yes, see Section 4.5.
- Can we solve byzantine agreement (or at least consensus) in less than $f + 1$ rounds?

4.4 Lower Bound on Number of Rounds

Theorem 4.20. *A synchronous algorithm solving consensus in the presence of f crashing nodes needs at least $f + 1$ rounds, if nodes decide for the minimum seen value.*

Proof. Let us assume (for the sake of contradiction) that some algorithm A solves consensus in f rounds. Some node u_1 has the smallest input value x, but in the first round u_1 can send its information (including information about its value x) to only some other node u_2 before u_1 crashes. Unfortunately, in the second round, the only witness u_2 of x also sends x to exactly one other node u_3 before u_2 crashes. This will be repeated, so in round f only node u_{f+1} knows about the smallest value x. As the algorithm terminates in round f, node u_{f+1} will decide on value x, all other surviving (correct) nodes will decide on values larger than x. □

Remarks:

- A general proof without the restriction to decide for the minimum value exists as well.
- Since byzantine nodes can also just crash, this lower bound also holds for byzantine agreement, so Algorithm 4.14 has an asymptotically optimal runtime.
- So far all our byzantine agreement algorithms assume the synchronous model. Can byzantine agreement be solved in the asynchronous model?

4.5 Asynchronous Byzantine Agreement

Algorithm 4.21 Asynchronous Byzantine Agreement (Ben-Or, for $f < n/10$)

```
 1: $x_u \in \{0,1\}$            ◁ input bit
 2: round = 1                    ◁ round
 3: while true do
 4:    Broadcast propose($x_u$,round)
 5:    Wait until $n - f$ propose messages of current round arrived
 6:    if at least $n/2 + 3f + 1$ propose messages contain same value $x$ then
 7:       Broadcast propose($x$,round + 1)
 8:       Decide for $x$ and terminate
 9:    else if at least $n/2 + f + 1$ propose messages contain same value $x$ then
10:       $x_u = x$
11:    else
12:       choose $x_u$ randomly, with $Pr[x_u = 0] = Pr[x_u = 1] = 1/2$
13:    end if
14:    round = round + 1
15: end while
```

Lemma 4.22. *Let a correct node choose value x in Line 10, then no other correct node chooses value $y \neq x$ in Line 10.*

Proof. For the sake of contradiction, assume that both 0 and 1 are chosen in Line 10. This means that both 0 and 1 had been proposed by at least $n/2 + 1$ out of $n - f$ correct nodes. In other words, we have a total of at least $2 \cdot n/2 + 2 = n + 2 > n - f$ correct nodes. Contradiction! □

Theorem 4.23. *Algorithm 4.21 solves binary byzantine agreement as in Definition 4.2 for up to $f < n/10$ byzantine nodes.*

Proof. First note that it is not a problem to wait for $n - f$ propose messages in Line 5, since at most f nodes are byzantine. If all correct nodes have the same input value x, then all (except the f byzantine nodes) will propose the same value x. Thus, every node receives at least $n - 2f$ propose messages containing x. Observe that for $f < n/10$, we get $n - 2f > n/2 + 3f$ and the nodes will decide on x in the first round already. We have established all-same validity! If the correct nodes have different (binary) input values, the validity condition becomes trivial as any result is fine.

What about agreement? Let u be the first node to decide on value x (in Line 8). Due to asynchrony, another node v received messages from a different subset of the nodes, however, at most f senders may be different. Taking into account that byzantine nodes may lie (send different propose messages to different nodes), f additional propose messages received by v may differ from those received by u. Since node u had at least $n/2 + 3f + 1$ propose messages with value x, node v has at least $n/2 + f + 1$ propose messages with value x. Hence every correct node will propose x in the next round and then decide on x.

So we only need to worry about termination: We have already seen that as soon as one correct node terminates (Line 8) everybody terminates in the next round. So what are the chances that some node u terminates in Line 8? Well, we can hope that all correct nodes randomly propose the same value (in Line 12). Maybe there are some nodes not choosing randomly (entering Line 10 instead of 12), but according to Lemma 4.22 they will all propose the same.

Thus, at worst all $n-f$ correct nodes need to randomly choose the same bit, which happens with probability $2^{-(n-f)+1}$. If so, all correct nodes will send the same propose message, and the algorithm terminates. So the expected running time is exponential in the number of nodes n in the worst case. □

Remarks:

- This Algorithm is a proof of concept that asynchronous byzantine agreement can be achieved. Unfortunately this algorithm is not useful in practice, because of its runtime.

- Note that for $f \in O(\sqrt{n})$, the probability for some node to terminate in Line 8 is greater than some positive constant. Thus, Algorithm 4.21 terminates within expected constant number of rounds for small values of f.

- Local coinflips are responsible for the slow runtime of Algorithm 4.21 and 3.15. Is there a simple way to replace the local coinflips by randomness that does not cause exponential runtime?

4.6 Random Oracle and Bitstring

Definition 4.24 (Random Oracle)**.** *A random oracle is a trusted (non-byzantine) random source which can generate random values.*

Algorithm 4.25 Algorithm 4.21 with a Magic Random Oracle

```
Replace Line 12 in Algorithm 4.21 by
return $c_i$, where $c_i$ is $i$th random bit by oracle
```

Remarks:

- Algorithm 4.25, as well as the following Algorithms 4.28 and 5.23, will be called in Line 12 of Algorithm 4.21. So instead of every node throwing a local coin (and hoping that they all show the same), the nodes will base their random decision on the proposed algorithm.

Theorem 4.26. *Algorithm 4.25 plugged into Algorithm 4.21 solves asynchronous byzantine agreement in expected constant number of rounds.*

Proof. If there is a large majority for one of the input values in the system, all nodes will decide within two rounds since Algorithm 4.21 satisfies all-same-validity; the coin is not even used.

If there is no significant majority for any of the input values at the beginning of algorithm 4.21, all correct nodes will run Algorithm 4.25. Therefore, they will set their new value to the bit given by the random oracle and terminate in the following round.

If neither of the above cases holds, some of the nodes see an $n/2 + f + 1$ majority for one of the input values, while other nodes rely on the oracle. With probability $1/2$, the value of the oracle will coincide with the deterministic majority value of the other nodes. Therefore, with probability $1/2$, the nodes will terminate in the following round. The expected number of rounds for termination in this case is 3. □

Remarks:

- Unfortunately, random oracles are a bit like pink fluffy unicorns: they do not really exist in the real world. Can we fix that?

Definition 4.27 (Random Bitstring)**.** *A **random bitstring** is a string of random binary values, known to all participating nodes when starting a protocol.*

Algorithm 4.28 Algorithm 4.21 with Random Bitstring

```
Replace Line 12 in Algorithm 4.21 by
return $b_i$, where $b_i$ is $i$th bit in common random bitstring
```

Remarks:

- But is such a precomputed bitstring really random enough? We should be worried because of Theorem 3.14.

Theorem 4.29. *If the scheduling is worst-case, Algorithm 4.28 plugged into Algorithm 4.21 does not terminate.*

Proof. We start Algorithm 4.28 with the following input: $n/2 + f + 1$ nodes have input value 1, and $n/2 - f - 1$ nodes have input value 0. Assume w.l.o.g. that the first bit of the random bitstring is 0.

If the second random bit in the bitstring is also 0, then a worst-case scheduler will let $n/2 + f + 1$ nodes see all $n/2 + f + 1$ values 1, these will therefore deterministically choose the value 1 as their new value. Because of scheduling (or byzantine nodes), the remaining $n/2 - f - 1$ nodes receive strictly less than

$n/2 + f + 1$ values 1 and therefore have to rely on the value of the shared coin, which is 0. The nodes will not come to a decision in this round. Moreover, we have created the very same distribution of values for the next round (which has also random bit 0).

If the second random bit in the bitstring is 1, then a worst-case scheduler can let $n/2 - f - 1$ nodes see all $n/2 + f + 1$ values 1, and therefore deterministically choose the value 1 as their new value. Because of scheduling (or byzantine nodes), the remaining $n/2 + f + 1$ nodes receive strictly less than $n/2 + f + 1$ values 1 and therefore have to rely on the value of the shared coin, which is 0. The nodes will not decide in this round. And we have created the symmetric situation for input value 1 that is coming in the next round.

So if the current and the next random bit are known, worst-case scheduling will keep the system in one of two symmetric states that never decide.

□

Remarks:

- Theorem 4.29 shows that a worst-case scheduler cannot be allowed to know the random bits of the future.
- Note that in the proof of Theorem 4.29 we did not even use any byzantine nodes. Just bad scheduling was enough to prevent termination.

Chapter Notes

The project which started the study of byzantine failures was called SIFT and was founded by NASA [WLG$^+$78], and the research regarding byzantine agreement started to get significant attention with the results by Pease, Shostak, and Lamport [PSL80, LSP82]. In [PSL80] they presented the generalized version of Algorithm 4.9 and also showed that byzantine agreement is unsolvable for $n \leq 3f$. The algorithm presented in that paper is nowadays called *Exponential Information Gathering (EIG)*, due to the exponential size of the messages.

There are many algorithms for the byzantine agreement problem. For example the Queen Algorithm [BG89] which has a better runtime than the King algorithm [BGP89], but tolerates less failures. That byzantine agreement requires at least $f + 1$ many rounds was shown by Dolev and Strong [DS83], based on a more complicated proof from Fischer and Lynch [FL82].

While many algorithms for the synchronous model have been around for a long time, the asynchronous model is a lot harder. The only results were by Ben-Or and Bracha. Ben-Or [Ben83] was able to tolerate $f < n/5$. Bracha [BT85] improved this tolerance to $f < n/3$.

Nearly all developed algorithms only satisfy all-same validity. There are a few exceptions, e.g., correct-input validity [FG03], available if the initial values are from a finite domain, median validity [SW15, MW18, DGM$^+$11] if the input values are orderable, or values inside the convex hull of all correct input values [VG13, MH13, MHVG15] if the input is multidimensional.

Before the term *byzantine* was coined, the terms Albanian Generals or Chinese Generals were used in order to describe malicious behavior. When the involved researchers met people from these countries they moved – for obvious reasons – to the historic term byzantine [LSP82].

Hat tip to Peter Robinson for noting how to improve Algorithm 4.9 to all-same validity. This chapter was written in collaboration with Barbara Keller.

Bibliography

[Ben83] Michael Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30. ACM, 1983.

[BG89] Piotr Berman and Juan A Garay. *Asymptotically optimal distributed consensus*. Springer, 1989.

[BGP89] Piotr Berman, Juan A. Garay, and Kenneth J. Perry. Towards optimal distributed consensus (extended abstract). In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 410–415, 1989.

[BT85] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM (JACM)*, 32(4):824–840, 1985.

[DGM$^+$11] Benjamin Doerr, Leslie Ann Goldberg, Lorenz Minder, Thomas Sauerwald, and Christian Scheideler. Stabilizing Consensus with the Power of Two Choices. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA, June 2011.

[DS83] Danny Dolev and H. Raymond Strong. Authenticated algorithms for byzantine agreement. *SIAM Journal on Computing*, 12(4):656–666, 1983.

[FG03] Matthias Fitzi and Juan A Garay. Efficient player-optimal protocols for strong and differential consensus. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 211–220. ACM, 2003.

[FL82] Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. 14(4):183–186, June 1982.

[LSP82] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[MH13] Hammurabi Mendes and Maurice Herlihy. Multidimensional Approximate Agreement in Byzantine Asynchronous Systems. In *Proceedings of the Forty-fifth Annual ACM Symposium on Theory of Computing*, STOC, June 2013.

[MHVG15] Hammurabi Mendes, Maurice Herlihy, Nitin Vaidya, and Vijay K. Garg. Multidimensional agreement in Byzantine systems. *Distributed Computing*, 28(6):423–441, January 2015.

[MW18] Darya Melnyk and Roger Wattenhofer. Byzantine Agreement with Interval Validity. In *37th Annual IEEE International Symposium on Reliable Distributed Systems (SRDS), Salvador, Bahia, Brazil*, October 2018.

[PSL80] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[SW15] David Stolz and Roger Wattenhofer. Byzantine Agreement with Median Validity. In *19th International Conference on Priniciples of Distributed Systems (OPODIS), Rennes, France*, 2015.

[VG13] Nitin H. Vaidya and Vijay K. Garg. Byzantine Vector Consensus in Complete Graphs. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC, July 2013.

[WLG$^+$78] John H. Wensley, Leslie Lamport, Jack Goldberg, Milton W. Green, Karl N. Levitt, P. M. Melliar-Smith, Robert E. Shostak, and Charles B. Weinstock. Sift: Design and analysis of a fault-tolerant computer for aircraft control. In *Proceedings of the IEEE*, pages 1240–1255, 1978.

Chapter 5

Broadcast & Shared Coins

In Chapter 4 we have developed a fast solution for synchronous byzantine agreement (Algorithm 4.14), yet our *asynchronous* byzantine agreement solution (Algorithm 4.21) is still awfully slow. Some simple methods to speed up the algorithms did not work, mostly due to unrealistic assumptions. Can we at least solve asynchronous (assuming worst-case scheduling) *consensus* if we have crash failures? Possibly based on some advanced communication methods?

5.1 Shared Coin on a Blackboard

Definition 5.1 (Shared Coin)**.** *A* ***shared coin*** *is a binary random variable shared among all nodes. It is* 0 *for all nodes with constant probability and* 1 *for all nodes with constant probability. The shared coin is allowed to fail (be* 0 *for some nodes and* 1 *for other nodes) with constant probability.*

Remarks:

- In Chapter 3, we have already seen a shared coin in Algorithm 3.22. For that shared coin, we implicitly assumed that message scheduling was random.
- Worst-case scheduling is an issue that we have only briefly considered so far, in particular, to show that the random bitstring does not help to speed up Algorithm 4.21.
- What if scheduling is worst-case in Algorithm 3.22?

Lemma 5.2. *Algorithm 3.22 has exponential expected running time under worst-case scheduling.*

Proof. In Algorithm 3.22, worst-case scheduling may hide up to f rare zero coinflips. In order to receive a zero as the outcome of the shared coin, the nodes need to generate at least $f+1$ zeros. The probability for this to happen is $(1/n)^{f+1}$, which is exponentially small for $f \in \Omega(n)$. In other words, with worst-case scheduling, with probability $1-(1/n)^{f+1}$ the shared coin will be 1. The worst-case scheduler must make sure that some nodes will always deterministically go for 0, and the algorithm needs n^{f+1} rounds until it terminates. □

Definition 5.3 (Blackboard Model)**.** *The* ***blackboard*** *is a trusted authority which supports two operations. A node can* ***write*** *its message to the blackboard and a node can* ***read*** *all the values that have been written to the blackboard so far.*

Remarks:

- We assume that the nodes cannot reconstruct the order in which the messages are written to the blackboard since the system is asynchronous.

Algorithm 5.4 Crash-Resilient Shared Coin with Blackboard (for node u)

```
while true do
  Choose new local coin $c_u = +1$ with probability $1/2$, else $c_u = -1$
  Write $c_u$ to the blackboard
  Set $C$ = Read all coinflips on the blackboard
  if $|C| \geq n^2$ then
    return sign(sum(C))
  end if
end while
```

Remarks:

- In Algorithm 5.4 the outcome of a coinflip is -1 or $+1$ instead of 0 or 1 because it simplifies the analysis, i.e., "$-1 \approx 0$".

- The *sign* function is used for the decision values. The sign function returns $+1$ if the sum of all coinflips in C is positive, and -1 if it is negative.

- The algorithm is unusual compared to other asynchronous algorithms we have dealt with so far. So far we often waited for $n - f$ messages from other nodes. In Algorithm 5.4, a single node can single-handedly generate all n^2 coinflips, without waiting.

- If a node does not need to wait for other nodes, we call the algorithm *wait-free.*

- Many similar definitions beyond wait-free exist: lock-free, deadlock-free, starvation-free, and generally non-blocking algorithms.

Theorem 5.5 (Central Limit Theorem)**.** *Let $\{X_1, X_2, \ldots, X_N\}$ be a sequence of independent random variables with $Pr[X_i = -1] = Pr[X_i = 1] = 1/2$ for all $i = 1, \ldots, N$. Then for every positive real number z,*

$$\lim_{N\to\infty} \Pr\left[\sum_{i=1}^{N} X_i \geq z\sqrt{N}\right] = 1 - \Phi(z) > \frac{1}{\sqrt{2\pi}}\frac{z}{z^2+1}e^{-z^2/2},$$

where $\Phi(z)$ is the cumulative distribution function of the standard normal distribution evaluated at z.

Theorem 5.6. *Algorithm 5.4 implements a polynomial shared coin.*

Proof. Each node in the algorithm terminates once at least n^2 coinflips are written to the blackboard. Before terminating, nodes may write one additional coinflip. Therefore, every node decides after reading at least n^2 and at most $n^2 + n - 1$ coinflips. The power of the adversary lies in the fact that it can prevent $n - 1$ nodes from writing their coinflips to the blackboard by delaying their writes. Here, we will consider an even stronger adversary that can hide up to n coinflips which were written on the blackboard.

We need to show that both outcomes for the shared coin ($+1$ or -1 in Line 6) will occur with constant probability, as in Definition 5.1. Let X be the sum of all coinflips that are visible to every node. Since some of the nodes might read n more values from the blackboard than others, the nodes cannot be prevented from deciding if $|X| > n$. By applying Theorem 5.5 with $N = n^2$ and $z = 1$, we get:

$$Pr(X \leq -n) = Pr(X \geq n) = 1 - \Phi(1) > 0.15.$$

□

Lemma 5.7. *Algorithm 5.4 uses n^2 coinflips, which is optimal in this model.*

Proof. The proof for showing quadratic lower bound makes use of configurations that are indistinguishable to all nodes, similar to Theorem 3.14. It requires involved stochastic methods and we therefore will only sketch the idea of where the n^2 comes from.

The basic idea follows from Theorem 5.5. The standard deviation of the sum of n^2 coinflips is n. The central limit theorem tells us that with constant probability the sum of the coinflips will be only a constant factor away from the standard deviation. As we showed in Theorem 5.6, this is large enough to disarm a worst-case scheduler. However, with much less than n^2 coinflips, a worst-case scheduler is still too powerful. If it sees a positive sum forming on the blackboard, it delays messages trying to write $+1$ in order to turn the sum temporarily negative, so the nodes finishing first see a negative sum, and the delayed nodes see a positive sum. □

Remarks:

- Algorithm 5.4 cannot tolerate even one byzantine failure: assume the byzantine node generates all the n^2 coinflips in every round due to worst-case scheduling. Then this byzantine node can make sure that its coinflips always sum up to a value larger than n, thus making the outcome -1 impossible.

- In Algorithm 5.4, we assume that the blackboard is a trusted central authority. Like the random oracle of Definition 4.24, assuming a blackboard does not seem practical. However, fortunately, we can use advanced broadcast methods in order to implement something like a blackboard with just messages.

5.2 Broadcast Abstractions

Definition 5.8 (Accept)**.** *A message received by a node v is called **accepted** if node v can consider this message for its computation.*

Definition 5.9 (Best-Effort Broadcast)**.** ***Best-effort broadcast** ensures that a message that is sent from a correct node u to another correct node v will eventually be received and accepted by v.*

Remarks:

- Note that best-effort broadcast is equivalent to the simple broadcast primitive that we have used so far.
- Reliable broadcast is a stronger paradigm which implies that byzantine nodes cannot send different values to different nodes. Such behavior will be detected.

Definition 5.10 (Reliable Broadcast)**.** ***Reliable broadcast** ensures that the nodes eventually agree on all accepted messages. That is, if a correct node v considers message m as accepted, then every other node will eventually consider message m as accepted.*

Algorithm 5.11 Asynchronous Reliable Broadcast (code for node u)

```
1: Broadcast own message msg($u$)
2: upon receiving msg($v$) from $v$ or echo($w$, msg($v$)) from $n - 2f$ nodes $w$:
3:     Broadcast echo($u$, msg($v$))
4: end upon
5: upon receiving echo($w$, msg($v$)) from $n - f$ nodes $w$:
6:     Accept msg($v$)
7: end upon
```

Theorem 5.12. *Algorithm 5.11 satisfies the following properties:*

1. *If a correct node broadcasts a message reliably, it will eventually be accepted by every other correct node.*
2. *If a correct node has not broadcast a message, it will not be accepted by any other correct node.*
3. *If a correct node accepts a message, it will be eventually accepted by every correct node*

This algorithm can tolerate $f < n/3$ byzantine nodes or $f < n/2$ crash failures.

Proof. We start with the first property. Assume a correct node broadcasts a message msg(v), then every correct node will receive msg(v) eventually. In Line 3, every correct node (including the originator of the message) will echo the message and, eventually, every correct node will receive at least $n - f$ echoes, thus accepting msg(v).

The second property holds for the case with crash failures, as all correct nodes follow the algorithm. In the byzantine case, byzantine nodes are not able to forge an incorrect sender address, see Definition 4.1. Instead, they can echo messages from correct nodes with a wrong input value. If all byzantine nodes echo a message that has not been broadcast by a correct node, each correct node will receive at most $f < n - 2f$ echo messages and thus no correct node will accept such a message.

For the third property, assume that some message originated from a byzantine node b, or a node b that has crashed in the process of sending its message. If a correct node accepted message $\texttt{msg}(b)$, this node must have received at least $n - f$ echoes for this message in Line 5. If at most f nodes are faulty, at least $n-2f$ correct nodes must have broadcast an echo message for $\texttt{msg}(b)$. Therefore, every correct node will receive these $n-2f$ echoes eventually and will broadcast an echo. Finally, all $n - f$ correct nodes will have broadcast an echo for $\texttt{msg}(b)$ and every correct node will accept $\texttt{msg}(b)$. □

Remarks:

- Algorithm 5.11 does not solve consensus according to Definition 3.1. It only makes sure that all messages of correct nodes will be accepted *eventually*. For correct nodes, this corresponds to sending and receiving messages in the asynchronous time model (Model 3.2).
- The algorithm has a linear message overhead since every node again broadcasts every message.
- Note that byzantine nodes can issue arbitrarily many messages. This may be a problem for protocols where each node is only allowed to send one message (per round). Can we fix this, for instance with sequence numbers?

Definition 5.13 (FIFO Reliable Broadcast)**.** *The **FIFO (reliable) broadcast** defines an order in which the messages are accepted in the system. If a node u broadcasts message m_1 before m_2, then any node v will accept message m_1 before m_2.*

Algorithm 5.14 FIFO Reliable Broadcast (code for node u)

```
Broadcast own round $r$ message $\texttt{msg}(u, r)$
upon receiving first message $\texttt{msg}(v, r)$ from node $v$ for round $r$ or $n - 2f$
    $\texttt{echo}(w, \text{msg}(v, r))$ messages:
  Broadcast $\texttt{echo}(u, \text{msg}(v, r))$
end upon
upon receiving $\texttt{echo}(w, \text{msg}(v, r))$ from $n - f$ nodes $w$:
  if accepted $\texttt{msg}(v, r - 1)$ then
    Accept $\texttt{msg}(v, r)$
  end if
end upon
```

Theorem 5.15. *Algorithm 5.14 satisfies the properties of Theorem 5.12. Additionally, Algorithm 5.14 makes sure that no two messages* $\texttt{msg}(v, r)$ *and* $\texttt{msg'}(v, r)$ *are accepted from the same node. It can tolerate $f < n/5$ byzantine nodes or $f < n/2$ crash failures.*

Proof. Just as reliable broadcast, Algorithm 5.14 satisfies the three properties of Theorem 5.12 by simply following the flow of messages of a correct node. It remains to show that at most one message will be accepted from some node v in round r. In the crash failure case, this property holds because all nodes follow the algorithm and therefore send at most one message in a round. For the byzantine case, assume some correct node u has accepted `msg`(v, r) in Line 7. This node must have received $n-f$ ECHO for this message, $n-2f$ of which were sent from the correct nodes. At least $n-2f-f = n-3f$ of those messages are sent for the first time by correct nodes. Now, assume for contradiction that another correct node accepts `msg'`(v, r). Similarly, $n-3f$ of those messages are sent for the first time by correct nodes. So, we have $n-3f+n-3f > n-f$ (for $f < n/5$) correct nodes sent ECHO for the first time. A contradiction.

□

Definition 5.16 (Atomic Broadcast)**.** ***Atomic broadcast*** *makes sure that all messages are received in the same order by every node. That is, for any pair of nodes u, v, and for any two messages m_1 and m_2, node u receives m_1 before m_2 if and only if node v receives m_1 before m_2.*

Remarks:

- Definition 5.16 is equivalent to Definition 2.8, i.e., atomic broadcast = state replication.
- Now we have all the tools to finally solve asynchronous consensus.

5.3 Blackboard with Message Passing

Algorithm 5.17 Crash-Resilient Shared Coin (code for node u)

```
1: while true do
2:    Choose local coin $c_u = +1$ with probability $1/2$, else $c_u = -1$
3:    FIFO-broadcast coin($c_u, r$) to all nodes
4:    Save all received coins coin($c_v, r$) in a set $C_u$
5:    Wait until accepted own coin($c_u$)
6:    Request $C_v$ from $n - f$ nodes $v$, and add newly seen coins to $C_u$
7:    if $|C_u| \geq n^2$ then
8:       return sign(sum($C_u$))
9:    end if
10: end while
```

Theorem 5.18. *Algorithm 5.17 solves asynchronous binary agreement for $f < n/2$ crash failures.*

Proof. The upper bound for the number of crash failures results from the upper bound in 5.15. The idea of this algorithm is to simulate the read and write operations from Algorithm 5.4.

Line 3 simulates a read operation: by accepting the own coinflip, a node verifies that $n-f$ correct nodes have received its most recent generated coinflip `coin`(c_u, r). At least $n-2f > 1$ of these nodes will never crash and the value therefore can be considered as stored on the blackboard. While a value is not accepted and therefore not stored, node u will not generate new coinflips. Therefore, at any point of the algorithm, there is at most n additional generated coinflips next to the accepted coins.

Line 6 of the algorithm corresponds to a read operation. A node reads a value by requesting C_v from at least $n - f$ nodes v. Assume that for a coinflip $\texttt{coin}(c_u, r)$, f nodes that participated in the FIFO broadcast of this message have crashed. When requesting $n - f$ sets of coinflips, there will be at least $(n - 2f) + (n - f) - (n - f) = n - 2f > 1$ sets among the requested ones containing $\texttt{coin}(c_u, r)$. Therefore, a node will always read all values that were accepted so far.

This shows that the read and write operations are equivalent to the same operations in Algorithm 5.4. Assume now that some correct node has terminated after reading n^2 coinflips. Since each node reads the stored coinflips before generating a new one in the next round, there will be at most n additional coins accepted by any other node before termination. This setting is equivalent to Theorem 5.6 and the rest of the analysis is therefore analogous to the analysis in that theorem. □

Remarks:

- So finally we can deal with worst-case crash failures *and* worst-case scheduling.
- But what about byzantine agreement? We need even more powerful methods!

5.4 Using Cryptography

Definition 5.19 (Threshold Secret Sharing)**.** *Let $t, n \in \mathbb{N}$ with $1 \leq t \leq n$. An algorithm that distributes a secret among n participants such that t participants need to collaborate to recover the secret is called a (t, n)**-threshold secret sharing** scheme.*

Definition 5.20 (Signature)**.** *Every node can **sign** its messages in a way that no other node can forge, thus nodes can reliably determine which node a signed message originated from. We denote a message x signed by node u with $\texttt{msg}(x)_u$.*

Algorithm 5.21 (t, n)-Threshold Secret Sharing

```
Input: A secret $s$, represented as a real number.

Secret distribution by dealer $d$

Generate $t-1$ random numbers $a_1, \ldots, a_{t-1} \in \mathbb{R}$
Obtain a polynomial $p$ of degree $t-1$ with $p(x) = s + a_1 x + \cdots + a_{t-1} x^{t-1}$
Generate $n$ distinct $x_1, \ldots, x_n \in \mathbb{R} \setminus \{0\}$
Distribute share $\texttt{msg}(x_1, p(x_1))_d$ to node $v_1$, ..., $\texttt{msg}(x_n, p(x_n))_d$ to node $v_n$

Secret recovery

Collect $t$ shares $\texttt{msg}(x_u, p(x_u))_d$ from at least $t$ nodes
Use Lagrange's interpolation formula to obtain $p(0) = s$
```

Remarks:

- Algorithm 5.21 relies on a trusted dealer, who broadcasts the secret shares to the nodes.
- Note that the communication between the dealer and the nodes must be private, i.e., a byzantine party cannot see the shares sent to the correct nodes.
- Using an $(f+1, n)$-threshold secret sharing scheme, we can encrypt messages in such a way that byzantine nodes alone cannot decrypt them.

Algorithm 5.22 Preprocessing Step for Algorithm 5.23 (code for dealer d)

```
1: According to Algorithm 5.21, choose polynomial $p$ of degree $f$
2: for $i = 1, \dots, n$ do
3:    Choose coinflip $c_i$, where $c_i = 0$ with probability 1/2, else $c_i = 1$
4:    Using Algorithm 5.21, generate $n$ shares $(x_1^i, p(x_1^i)), \dots, (x_n^i, p(x_n^i))$ for $c_i$
5: end for
6: Send shares $\texttt{msg}(x_u^1, p(x_u^1))_d, \dots, \texttt{msg}(x_u^n, p(x_u^n))_d$ to node $u$
```

Algorithm 5.23 Shared Coin using Secret Sharing (ith iteration)

```
1: Replace Line 12 in Algorithm 4.21 by
2: Request shares from at least $f + 1$ nodes
3: Using Algorithm 5.21, let $c_i$ be the value reconstructed from the shares
4: return $c_i$
```

Theorem 5.24. *Algorithm 4.21 together with Algorithm 5.22 and Algorithm 5.23 solves asynchronous byzantine agreement for $f < n/3$ in expected 3 number of rounds.*

Proof. In Line 2 of Algorithm 5.23, the nodes collect shares from $f+1$ nodes. Since a byzantine node cannot forge the signature of the dealer, it is restricted to either send its own share or decide to not send it at all. Therefore, each correct node will eventually be able to reconstruct secret c_i of round i correctly in Line 3 of the algorithm. The running time analysis follows then from the analysis of Theorem 4.26. □

Remarks:

-
- In Algorithm 5.22 we assume that the dealer generates the random bitstring. This assumption is not necessary if the communication between any pair of nodes is private: In a setup phase of the algorithm, each node can generate a local coinflip and broadcast the secret shares of its coinflip to all other nodes. The corresponding secret will only be revealed in a designated round of the algorithm, thus keeping the outcome of the coinflip secret to a byzantine adversary.

- We showed that cryptographic assumptions can speed up asynchronous byzantine agreement.
- Algorithm 4.21 can also be implemented in the synchronous setting.
- A randomized version of a synchronous byzantine agreement algorithm can improve on the lower bound of $t + 1$ rounds for the deterministic algorithms.

Definition 5.25 (Cryptographic Hash Function)**.** *A hash function hash* $: U \to S$ *is called **cryptographic**, if for a given* $z \in S$ *it is computationally hard to find an element* $x \in U$ *with hash*$(x) = z$*.*

Remarks:

- Popular hash functions used in cryptography include the Secure Hash Algorithm (SHA) and the Message-Digest Algorithm (MD).

Algorithm 5.26 Simple Synchronous Byzantine Shared Coin (for node u)

```
1: Each node has a public key that is known to all nodes.
2: Let $r$ be the current round of Algorithm 4.21
3: Broadcast $\texttt{msg}(r)_u$, i.e., round number $r$ signed by node $u$
4: Compute $h_v = \text{hash}(\text{msg}(r)_v)$ for all received messages $\texttt{msg}(r)_v$
5: Let $h_{min} = \min_v\ h_v$
6: return least significant bit of $h_{min}$
```

Remarks:

- In Algorithm 5.26, Line 3 each node can verify the correctness of the signed message using the public key.
- Just as in Algorithm 4.9, the decision value is the minimum of all received values. While the minimum value is received by all nodes after 2 rounds there, we can only guarantee to receive the minimum with constant probability in this algorithm.
- Hashing helps to restrict byzantine power, since a byzantine node cannot compute the smallest hash.

Theorem 5.27. *Algorithm 5.26 plugged into Algorithm 4.21 solves synchronous byzantine agreement in expected* 5 *rounds for up to* $f < n/10$ *byzantine failures.*

Proof. With probability $1/3$ the minimum hash value is generated by a byzantine node. In such a case, we can assume that not all correct nodes will receive the byzantine value and thus, different nodes might compute different values for the shared coin.

With probability $2/3$, the shared coin will be from a correct node, and with probability $1/2$ the value of the shared coin will correspond to the value which was deterministically chosen by some of the correct nodes. Therefore, with probability $1/3$ the nodes will reach consensus in the next iteration of Algorithm 4.21. Thus the expected number of rounds is 4 (expected number of rounds to be lucky in a round is 3 plus one more iteration to terminate). □

Chapter Notes

Asynchronous byzantine agreement is usually considered in one out of two communication models – shared memory or message passing. The first polynomial algorithm for the shared memory model that uses a shared coin was proposed by Aspnes and Herlihy [AH90] and required exchanging $O(n^4)$ messages in total. Algorithm 5.4 is also an implementation of the shared coin in the shared memory model and it requires exchanging $O(n^3)$ messages. This variant is due to Saks, Shavit and Woll [SSW91]. Bracha and Rachman [BR92] later reduced the number of messages exchanged to $O(n^2 \log n)$. The tight lower bound of $\Omega(n^2)$ on the number of coinflips was proposed by Attiya and Censor [AC08] and improved the first non-trivial lower bound of $\Omega(n^2/\log^2 n)$ by Aspnes [Asp98].

In the message passing model, the shared coin is usually implemented using reliable broadcast. Reliable broadcast was first proposed by Srikanth and Toueg [ST87] as a method to simulate authenticated broadcast. There is also another implementation which was proposed by Bracha [Bra87]. Today, a lot of variants of reliable broadcast exist, including FIFO broadcast [AAD05], which was considered in this chapter. A good overview over the broadcast routines is given by Cachin et al. [CGR14]. A possible way to reduce message complexity is by simulating the read and write commands [ABND95] as in Theorem **??**. The message complexity of this method is $O(n^3)$. Alistarh et al. [AAKS14] improved the number of exchanged messages to $O(n^2 \log^2 n)$ using a binary tree that restricts the number of communicating nodes according to the depth of the tree.

It remains an open question whether asynchronous byzantine agreement can be solved in the message passing model without cryptographic assumptions. If cryptographic assumptions are however used, byzantine agreement can be solved in expected constant number of rounds. Algorithm 5.22 presents the first implementation due to Rabin [Rab83] using threshold secret sharing. This algorithm relies on the fact that the dealer provides the random bitstring. Chor et al. [CGMA85] proposed the first algorithm where the nodes use verifiable secret sharing in order to generate random bits. Later work focuses on improving resilience [CR93] and practicability [CKS00]. Algorithm 5.26 by Micali [Mic18] shows that cryptographic assumptions can also help to improve the running time in the synchronous model.

This chapter was written in collaboration with Darya Melnyk.

Bibliography

[AAD05] Ittai Abraham, Yonatan Amit, and Danny Dolev. Optimal resilience asynchronous approximate agreement. In *Proceedings of the 8th International Conference on Principles of Distributed Systems*, OPODIS'04, pages 229–239, Berlin, Heidelberg, 2005. Springer-Verlag.

[AAKS14] Dan Alistarh, James Aspnes, Valerie King, and Jared Saia. Communication-efficient randomized consensus. In Fabian Kuhn, editor, *Distributed Computing*, pages 61–75, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[ABND95] Hagit Attiya, Amotz Bar-Noy, and Danny Dolev. Sharing memory robustly in message-passing systems. *J. ACM*, 42(1):124–142, January 1995.

[AC08] Hagit Attiya and Keren Censor. Tight bounds for asynchronous randomized consensus. *J. ACM*, 55(5):20:1–20:26, November 2008.

[AH90] James Aspnes and Maurice Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441 – 461, 1990.

[Asp98] James Aspnes. Lower bounds for distributed coin-flipping and randomized consensus. *J. ACM*, 45(3):415–450, May 1998.

[BR92] Gabriel Bracha and Ophir Rachman. Randomized consensus in expected $o(n^2 logn)$ operations. In *Proceedings of the 5th International Workshop on Distributed Algorithms*, WDAG '91, pages 143–150, Berlin, Heidelberg, 1992. Springer-Verlag.

[Bra87] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130 – 143, 1987.

[CGMA85] B. Chor, S. Goldwasser, S. Micali, and B. Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pages 383–395, Oct 1985.

[CGR14] Christian Cachin, Rachid Guerraoui, and Lus Rodrigues. *Introduction to Reliable and Secure Distributed Programming.* Springer Publishing Company, Incorporated, 2nd edition, 2014.

[CKS00] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *Journal of Cryptology*, 18:219–246, 2000.

[CR93] Ran Canetti and Tal Rabin. Fast asynchronous byzantine agreement with optimal resilience. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing*, STOC '93, pages 42–51, New York, NY, USA, 1993. ACM.

[Mic18] Silvio Micali. Byzantine agreement , made trivial. 2018.

[Rab83] M. O. Rabin. Randomized byzantine generals. In *24th Annual Symposium on Foundations of Computer Science (sfcs 1983)*, pages 403–409, Nov 1983.

[SSW91] Michael Saks, Nir Shavit, and Heather Woll. Optimal time randomized consensus – making resilient algorithms fast in practice. In *Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '91, pages 351–362, Philadelphia, PA, USA, 1991. Society for Industrial and Applied Mathematics.

[ST87] T. K. Srikanth and S. Toueg. Optimal Clock Synchronization. *Journal of the ACM*, 34:626–645, 1987.

Chapter 6

Consistency & Logical Time

You submit a comment on your favorite social media platform using your phone. The comment is immediately visible on the phone, but not on your laptop. Is this level of consistency acceptable?

6.1 Consistency Models

Definition 6.1 (Object)**.** *An* ***object*** *is a variable or a data structure storing information.*

Remarks:

- Object is a general term for any entity that can be modified, like a queue, stack, memory slot, file system, etc.

Definition 6.2 (Operation)**.** *An* ***operation*** *f accesses or manipulates an object. The operation f starts at wall-clock time f_* and ends at wall-clock time $f_\dagger$.*

Remarks:

- An operation can be as simple as extracting an element from a data structure, but an operation may also be more complex, like fetching an element, modifying it and storing it again.

- If $f_\dagger < g_*$, we simply write $f < g$.

Definition 6.3 (Execution)**.** *An* ***execution*** *E is a set of operations on one or multiple objects that are executed by a set of nodes.*

Definition 6.4 (Sequential Execution)**.** *An execution restricted to a single node is a* ***sequential execution****. All operations are executed sequentially, which means that no two operations f and g are concurrent, i.e., we have $f < g$ or $g < f$.*

Remarks:

- Arguing about correctness of executions is generally difficult. Even in the much simpler case of sequential executions there are advanced tools to argue about correctness, such as model checking or formal methods. We consider a sequential execution to be correct if the operations manipulate the objects as expected, e.g. if we add 2+2 we want to see a result of 4.

Definition 6.5 (Semantic Equivalence)**.** *Two executions are* ***semantically equivalent*** *if they contain exactly the same operations. Moreover, each pair of corresponding operations has the same effect in both executions.*

Remarks:

- For example, when dealing with a stack object, corresponding `pop` operations in two different semantically equivalent executions must yield the same element of the stack.
- In general, the notion of semantic equivalence is non-trivial and dependent on the type of the object.

Definition 6.6 (Linearizability)**.** *An execution E is called* ***linearizable*** *(or atomically consistent), if there is a sequence of operations (sequential execution) S such that:*

- *S is correct and semantically equivalent to E.*
- *Whenever $f < g$ for two operations f, g in E, then also $f < g$ in S.*

Definition 6.7. *A* ***linearization point*** *of operation f is some $f_\bullet \in [f_*, f_\dagger]$.*

Lemma 6.8. *An execution E is linearizable if and only if there exist linearization points such that the sequential execution S that results in ordering the operations according to those linearization points is semantically equivalent to E.*

Proof. Let f and g be two operations in E with $f_\dagger < g_*$. Then by definition of linearization points we also have $f_\bullet < g_\bullet$ and therefore $f < g$ in S. □

Definition 6.9 (Sequential Consistency)**.** *An execution E is called* ***sequentially consistent****, if there is a sequence of operations S such that:*

- *S is correct and semantically equivalent to E.*
- *Whenever $f < g$ for two operations f, g* **on the same node** *in E, then also $f < g$ in S.*

Lemma 6.10. *Every linearizable execution is also sequentially consistent, i.e., linearizability $\implies$ sequential consistency.*

Proof. Since linearizability (order of operations *on any* nodes must be respected) is stricter than sequential consistency (only order of operations *on the same node* must be respected), the lemma follows immediately. □

Definition 6.11 (Quiescent Consistency). *An execution E is called* ***quiescently consistent****, if there is a sequence of operations S such that:*

- *S is correct and semantically equivalent to E.*
- *Let t be some quiescent point, i.e., for all operations f we have $f_\dagger < t$ or $f_* > t$. Then for every t and every pair of operations g, h with $g_\dagger < t$ and $h_* > t$ we also have $g < h$ in S.*

Lemma 6.12. *Every linearizable execution is also quiescently consistent, i.e., linearizability $\implies$ quiescent consistency.*

Proof. Let E be the original execution and S be the semantically equivalent sequential execution. Let t be a quiescent point and consider two operations g, h with $g_\dagger < t < h_*$. Then we have $g < h$ in S. This is also guaranteed by linearizability since $g_\dagger < t < h_*$ implies $g < h$. □

Lemma 6.13. *Sequentially consistent and quiescent consistency do not imply one another.*

Proof. There are executions that are sequentially consistent but not quiescently consistent. An object initially has value 2. We apply two operations to this object: *inc* (increment the object by 1) and *double* (multiply the object by 2). Assume that $inc < double$, but *inc* and *double* are executed on different nodes. Then a result of 5 (first double, then inc) is sequentially consistent but not quiescently consistent.

There are executions that are quiescently consistent but not sequentially consistent. An object initially has value 2. Assume to have three operations on two nodes u and v. Node u calls first *inc* then *double*, node v calls *inc* once with $inc_*^v < inc_\dagger^u < double_*^u < inc_\dagger^v$. Since there is no quiescent point, quiescent consistency is okay with a sequential execution that doubles first, resulting in $((2 \cdot 2) + 1) + 1 = 6$. The sequential execution demands that $inc^u < double^u$, hence the result should be strictly larger than 6 (either 7 or 8). □

Definition 6.14. *A system or an implementation is called* ***linearizable*** *if it ensures that every possible execution is linearizable. Analogous definitions exist for sequential and quiescent consistency.*

Remarks:

- In the introductory social media example, a linearizable implementation would have to make sure that the comment is immediately visible on any device, as the *read* operation starts after the *write* operation finishes. If the system is only sequentially consistent, the comment does not need to be immediately visible on every device.

Definition 6.15 (restricted execution). *Let E be an execution involving operations on multiple objects. For some object o we let the* ***restricted execution*** *$E|o$ be the execution E filtered to only contain operations involving object o.*

Definition 6.16. *A consistency model is called* ***composable*** *if the following holds: If for every object o the restricted execution $E|o$ is consistent, then also E is consistent.*

Remarks:

- Composability enables to implement, verify and execute multiple concurrent objects independently.

Lemma 6.17. *Sequential consistency is not composable.*

Proof. We consider an execution E with two nodes u and v, which operate on two objects x and y initially set to 0. The operations are as follows: u_1 reads $x = 1$, u_2 writes $y := 1$, v_1 reads $y = 1$, v_2 writes $x := 1$ with $u_1 < u_2$ on node u and $v_1 < v_2$ on node v. It is clear that $E|x$ as well as $E|y$ are sequentially consistent as the write operations may be before the respective read operations. In contrast, execution E is *not* sequentially consistent: Neither u_1 nor v_1 can possibly be the initial operation in any correct semantically equivalent sequential execution S, as that would imply reading 1 when the variable is still 0. □

Theorem 6.18. *Linearizability is composable.*

Proof. Let E be an execution composed of multiple restricted executions $E|x$. For any object x there is a sequential execution $S|x$ that is semantically consistent to $E|x$ and in which the operations are ordered according to wall-clock-linearization points. Let S be the sequential execution ordered according to all linearization points of all executions $E|x$. S is semantically equivalent to E as $S|x$ is semantically equivalent to $E|x$ for all objects x and two object-disjoint executions cannot interfere. Furthermore, if $f_\dagger < g_*$ in E, then also $f_\bullet < g_\bullet$ in E and therefore also $f < g$ in S. □

6.2 Logical Clocks

To capture dependencies between nodes in an implementation, we can use logical clocks. These are supposed to respect the so-called happened-before relation.

Definition 6.19. *Let S_u be a sequence of operations on some node u and define "$\rightarrow$" to be the* ***happened-before relation*** *on $E := S_1 \cup \cdots \cup S_n$ that satisfies the following three conditions:*

1. *If a local operation f occurs before operation g on the same node $(f < g)$, then $f \rightarrow g$.*

2. *If f is a send operation of one node, and g is the corresponding receive operation of another node, then $f \rightarrow g$.*

3. *If f, g, h are operations such that $f \rightarrow g$ and $g \rightarrow h$ then also $f \rightarrow h$.*

Remarks:

- If for two distinct operations f, g neither $f \rightarrow g$ nor $g \rightarrow f$, then we also say f and g are *independent* and write $f \sim g$. Sequential computations are characterized by $\rightarrow$ being a total order, whereas the computation is entirely concurrent if no operations f, g with $f \rightarrow g$ exist.

Definition 6.20 (Happened-before consistency)**.** *An execution E is called* ***happened-before consistent****, if there is a sequence of operations S such that:*

- *S is correct and semantically equivalent to E.*
- *Whenever $f \to g$ for two operations f, g in E, then also $f < g$ in S.*

Lemma 6.21. *Happened-before consistency = sequential consistency.*

Proof. Both consistency models execute all operations of a single node in the sequential order. In addition, happened-before consistency also respects messages between nodes. However, messages are also ordered by sequential consistency because of semantic equivalence (a receive cannot be before the corresponding send). Finally, even though transitivity is defined more formally in happened-before consistency, also sequential consistency respects transitivity.

In addition, sequential consistency orders two operations o_u, o_v on two different nodes u, v if o_v can see a state change caused by o_u. Such a state change does not happen out of the blue, in practice some messages between u and v (maybe via "shared blackboard" or some other form of communication) will be involved to communicate the state change. □

Definition 6.22 (Logical clock)**.** *A logical clock is a family of functions c_u that map every operation $f \in E$ on node u to some logical time $c_u(f)$ such that the happened-before relation $\to$ is respected, i.e., for two operations g on node u and h on node v*

$$g \to h \implies c_u(g) < c_v(h).$$

Definition 6.23. *If it additionally holds that $c_u(g) < c_v(h) \implies g \to h$, then the clock is called a **strong logical clock**.*

Remarks:

- In algorithms we write c_u for the current logical time of node u.
- The simplest logical clock is the *Lamport clock*, given in Algorithm 6.24. Every message includes a timestamp, such that the receiving node may update its current logical time.

Algorithm 6.24 Lamport clock (code for node u)

```
Initialize $c_u := 0$.
Upon local operation: Increment current local time $c_u := c_u + 1$.
Upon send operation: Increment $c_u := c_u + 1$ and include $c_u$ as $T$ in message
Upon receive operation: Extract $T$ from message and update $c_u := \max(c_u, T) + 1$.
```

Theorem 6.25. *Lamport clocks are logical clocks.*

Proof. If for two operations f, g it holds that $f \to g$, then according to the definition three cases are possible.

1. If $f < g$ on the same node u, then $c_u(f) < c_u(g)$.
2. Let g be a receive operation on node v corresponding to some send operation f on another node u. We have $c_v(g) \geq T + 1 = c_u(f) + 1 > c_u(f)$.

3. Transitivity follows with $f \rightarrow g$ and $g \rightarrow h \Rightarrow f \rightarrow h$, and the first two cases.

□

Remarks:

- Lamport logical clocks are not strong logical clocks, which means we cannot completely reconstruct $\rightarrow$ from the family of clocks c_u.
- To achieve a strong logical clock, nodes also have to gather information about other clocks in the system, i.e., node u needs to have a idea of node v's clock, for every u, v. This is what *vector clocks* in Algorithm 6.26 do: Each node u stores its knowledge about other node's logical clocks in an n-dimensional vector c_u.

Algorithm 6.26 Vector clocks (code for node u)

```
Initialize $c_u[v] := 0$ for all other nodes $v$.
Upon local operation: Increment current local time $c_u[u] := c_u[u] + 1$.
Upon send operation: Increment $c_u[u] := c_u[u] + 1$ and include the whole
    vector $c_u$ as $d$ in message.
Upon receive operation: Extract vector $d$ from message and update $c_u[v] :=$
    $\max(d[v], c_u[v])$ for all entries $v$. Increment $c_u[u] := c_u[u] + 1$.
```

Theorem 6.27. *Define $c_u < c_v$ if and only if $c_u[w] \leq c_v[w]$ for all entries w, and $c_u[x] < c_v[x]$ for at least one entry x. Then the vector clocks are strong logical clocks.*

Proof. We are given two operations f, g, with operation f on node u, and operation g on node v, possibly $v = u$.

If we have $f \rightarrow g$, then there must be a happened-before-path of operations and messages from f to g. According to Algorithm 6.26, $c_v(g)$ must include at least the values of the vector $c_u(f)$, and the value $c_v(g)[v] > c_u(f)[v]$.

If we do not have $f \rightarrow g$, then $c_v(g)[u]$ cannot know about $c_u(f)[u]$, and hence $c_v(g)[u] < c_u(f)[u]$, since $c_u(f)[u]$ was incremented when executing f on node u.

□

Remarks:

- Usually the number of interacting nodes is small compared to the overall number of nodes. Therefore we do not need to send the full length clock vector, but only a vector containing the entries of the nodes that are actually communicating. This optimization is the called the *differential technique.*

Algorithm 6.28 Centralized Mutual Exclusion Algorithm

```
To access shared resource: Send request message to leader and wait for
  permission.
To release shared resource: Send release message to leader.
```

6.3 Application: Mutual Exclusion

When multiple nodes compete for exclusive access to a shared resource, we need a protocol which coordinates the order in which the resource gets assigned to the nodes. The most obvious algorithm is letting a leader node organize everything:

Remarks:

- An advantage of Algorithm 6.28 is its simplicity and low message overhead with 3 messages per access.

- An obvious disadvantage is that the leader is single point of failure and performance bottleneck. Assuming an asynchronous system, this protocol also does not achieve first come first serve fairness.

- We can solve these issues with a distributed algorithm using logical clocks:

Algorithm 6.29 Distributed Mutual Exclusion Algorithm

```
1: To access shared resource: Send message to all nodes containing the node
   ID and the current timestamp.
2: Upon received request message: If access to the same resource is needed
   and the own timestamp is lower than timestamp in received message, defer
   response. Otherwise send back a response.
3: Upon responses from all nodes received: enter critical section. Afterwards
   send deferred responses.
```

Remarks:

- The algorithm guarantees mutual exclusion without deadlocks or starvation of a requesting process.

- The number of messages per entry is $2(n-1)$, where n is the number of nodes in the system: $(n-1)$ requests and $(n-1)$ responses.

- There is no single point of failure. Yet, whenever a node crashes, it will not reply with a response and the requesting node waits forever. Even worse, the requesting process cannot determine if the silence is due to the other process currently accessing the shared resource or crashing. Can we fix this? Indeed: Change step 2 in Algorithm 6.29 such that upon receiving request there will always be an answer, either Denied or OK. This way crashes will be detected.

6.4 Consistent Snapshots

Definition 6.30 (cut)**.** *A **cut** is some prefix of a distributed execution. More precisely, if a cut contains an operation f on some node u, then it also contains all the preceding operations of u. The set of last operations on every node included in the cut is called the **frontier** of the cut. A cut C is called **consistent** if for every operation g in C with $f \to g$, C also contains f.*

Definition 6.31 (consistent snapshot)**.** *A **consistent snapshot** is a consistent cut C plus all messages in transit at the frontier of C.*

Remarks:

- In a consistent snapshot it is forbidden to see an effect without its cause.
- The number of possible consistent snapshots gives also information about the degree of concurrency of the system.
- One extreme is a sequential computation, where stopping one node halts the whole system. Let q_u be the number of operations on node $u \in \{1, \ldots, n\}$. Then the number of consistent snapshots (including the empty cut) in the sequential case is $\mu_s := 1 + q_1 + q_2 + \cdots + q_n$.
- One the other hand, in an entirely concurrent computation the nodes are not dependent on one another and therefore stopping one node does not impact others. The number of consistent snapshots in this case is $\mu_c := (1 + q_1) \cdot (1 + q_2) \cdots (1 + q_n)$.

Definition 6.32 (measure of concurrency)**.** *The concurrency measure of an execution $E = (S_1, \ldots, S_n)$ is defined as the ratio*

$$m(E) := \frac{\mu - \mu_s}{\mu_c - \mu_s},$$

where μ denotes the number of consistent snapshot of E.

Remarks:

- This measure of concurrency is normalized to $[0, 1]$.
- In order to evaluate the extent to which a computation is concurrent, we need to compute the number of consistent snapshots μ. This can be done via vector clocks.
- Imagine a bank having lots of accounts with transactions all over the world. The bank wants to make sure that at no point in time money gets created or destroyed. This is where consistent snapshots come in: They are supposed to capture the state of the system. Theoretically, we have already used snapshots when we discussed configurations in Definition 3.4:

Definition 6.33 (configuration)**.** *We say that a system is fully defined (at any point during the execution) by its **configuration**. The configuration includes the state of every node, and all messages that are in transit (sent but not yet received).*

Remarks:

- While a configuration describes the intractable state of a system at one point in time, a snapshot extracts all relevant tractable information of the systems state.

- One application of consistent snapshots is to check if certain invariants hold in a distributed setting. Other applications include distributed debugging or determining global states of a distributed system.

- In Algorithm 6.34 we assume that a node can record only its internal state and the messages it sends and receives. There is no common clock so it is not possible to just let each node record all information at precisely the same time.

Algorithm 6.34 Distributed Snapshot Algorithm

```
Initiator: Save local state, send a snap message to all other nodes and collect
  incoming states and messages of all other nodes.
All other nodes:
Upon receiving a snap message for the first time: send own state (before
  message) to the initiator and propagate snap by adding snap tag to future
  messages.
If afterwards receiving a message $m$ without snap tag: Forward $m$ to the
  initiator.
```

Theorem 6.35. *Algorithm 6.34 collects a consistent snapshot.*

Proof. Let C be the cut induced by the frontier of all states and messages forwarded to the initiator. For every node u, let t_u be the time when u gets the first snap message m (either by the initiator, or as a message tag). Then C contains all of u's operations before t_u, and none after t_u (also not the message m which arrives together with the tag at t_u).

Assume for the sake of contradiction we have operations f, g on nodes u, v respectively, with $f \to g$, $f \notin C$ and $g \in C$, hence $t_u \leq f$ and $g < t_v$. If $u = v$ we have $t_u \leq f < g < t_v = t_u$, which is a contradiction. On the other hand, if $u \neq v$: Since $t_u \leq f$ we know that all following send operations must have included the snap tag. Because of $f \to g$ we know there is a path of messages between f and g, all including the snap tag. So the snap tag must have been received by node v before or with operation g, hence $t_v \leq g$, which is a contradiction to $t_v > g$. □

Remarks:

- It may of course happen that a node u sends a message m before receiving the first snap message at time t_u (hence not containing the snap tag), and this message m is only received by node v after t_v. Such a message m will be reported by v, and is as such included in the consistent snapshot (as a message that was *in transit* during the snapshot).

6.5 Distributed Tracing

Definition 6.36 (Microservice Architecture)**.** *A **microservice architecture** refers to a system composed of loosely coupled services. These services communicate by various protocols and are either decentrally coordinated (also known as "choreography") or centrally ("orchestration").*

Remarks:

- There is no exact definition for microservices. A rule of thumb is that you should be able to program a microservice from scratch within two weeks.

- Microservices are the architecture of choice to implement a cloud based distributed system, as they allow for different technology stacks, often also simplifying scalability issues.

- In contrast to a monolithic architecture, debugging and optimizing get trickier as it is difficult to detect which component exactly is causing problems.

- Due to the often heterogeneous technology, a uniform debugging framework is not feasible.

- Tracing enables tracking the set of services which participate in some task, and their interactions.

Definition 6.37 (Span)**.** *A **span** s is a named and timed operation representing a contiguous sequence of operations on one node. A span s has a start time s_* and finish time $s_\dagger$.*

Remarks:

- Spans represent tasks, like a client submitting a request or a server processing this request. Spans often trigger several child spans or forwards the work to another service.

Definition 6.38 (Span Reference)**.** *A span may causally depend on other spans. The two possible relations are **ChildOf** and **FollowsFrom** references. In a ChildOf reference, the parent span depends on the result of the child (the parents asks the child and the child answers), and therefore parent and child span must overlap. In FollowsFrom references parent spans do not depend in any way on the result of their child spans (the parent just invokes the child).*

Definition 6.39 (Trace)**.** *A **trace** is a series-parallel directed acyclic graph representing the hierarchy of spans that are executed to serve some request. Edges are annotated by the type of the reference, either ChildOf or FollowsFrom.*

Algorithm 6.40 Inter-Service Tracing

```
Upon requesting another service: Inject information of current trace and
  span (IDs or timing information) into the request header.
Upon receiving request from another service: Extract trace and span infor-
  mation from the request header and create new span as child span.
```

Remarks:

- The advantage of using an open source definition like opentracing is that it is easy to replace a specific tracing by another one. This mitigates the lock-in effect that is often experienced when using some specific technology.
- Algorithm 6.40 shows what is needed if you want to trace requests to your system.

Remarks:

- All tracing information is collected and has to be sent to some tracing backend which stores the traces and usually provides a frontend to understand what is going on.
- Opentracing implementations are available for the most commonly used programming frameworks and can therefore be used for heterogeneous collections of microservices.

Chapter Notes

In his seminal work, Leslie Lamport came up with the happened-before relation and gave the first logical clock algorithm [Lam78]. This paper also laid the foundation for the theory of logical clocks. Fidge came some time later up with vector clocks [JF88]. An obvious drawback of vector clocks is the overhead caused by including the whole vector. Can we do better? In general, we cannot if we need strong logical clocks [CB91].

Lamport also introduced the algorithm for distributed snapshots, together with Chandy [CL85]. Besides this very basic algorithm, there exist several other algorithms, e.g., [LY87], [SK86].

Throughout the literature the definitions for, e.g., consistency or atomicity slightly differ. These concepts are studied in different communities, e.g., linearizability hails from the distributed systems community whereas the notion of serializability was first treated by the database community. As the two areas converged, the terminology got overloaded.

Our definitions for distributed tracing follow the OpenTracing API [1]. The opentracing API only gives high-level definitions of how a tracing system is supposed to work. Only the implementation specifies how it works internally.There are several systems that implement these generic definitions, like Uber's open source tracer called *Jaeger*, or *Zipkin*, which was first developed by Twitter. This technology is relevant for the growing number of companies that embrace

[1] http://opentracing.io/documentation/

a microservice architecture. Netflix for example has a growing number of over 1,000 microservices.

This chapter was written in collaboration with Julian Steger.

Bibliography

[CB91] Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Inf. Process. Lett.*, 39(1):11–16, July 1991.

[CL85] K Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. 3:63–75, 02 1985.

[JF88] Colin J. Fidge. Timestamps in message-passing systems that preserve partial ordering. 10:56–66, 02 1988.

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, jul 1978.

[LY87] Ten H. Lai and Tao H. Yang. On distributed snapshots. *Information Processing Letters*, 25(3):153 – 158, 1987.

[SK86] Madalene Spezialetti and Phil Kearns. Efficient distributed snapshots. In *ICDCS*, pages 382–388. IEEE Computer Society, 1986.

Chapter 7

Time, Clocks & GPS

> "A man with a clock knows what time it is – a man with two is never sure." (Segal's Law)

7.1 Time & Clocks

Definition 7.1 (Second)**.** *A* ***second*** *is the time that passes during 9,192,631,770 oscillation cycles of a caesium-133 atom.*

Remarks:

- This definition is a bit simplified. The official definition is given by the *Bureau International des Poids et Mesures.*
- Historically, a second was defined as one in 86,400 parts of a day, dividing the day into 24 hours, 60 minutes and 60 seconds.
- Since the duration of a day depends on the unsteady rotation cycle of the Earth, the novel oscillation-based definition has been adopted. Leap seconds are used to keep time synchronized to Earth's rotation.

Definition 7.2 (Wall-Clock Time)**.** *The* ***wall-clock time*** t^* *is the true time (a perfectly accurate clock would show).*

Definition 7.3 (Clock)**.** *A* ***clock*** *is a device which tracks and indicates time.*

Remarks:

- A clock's time t is a function of the wall-clock time t^*, i.e., $t = f(t^*)$. Ideally, $t = t^*$, but in reality there are often errors.

Definition 7.4 (Clock Error)**.** *The* ***clock error*** *or clock skew is the difference between two clocks, e.g.,* $t - t^*$ *or* $t - t'$*. In practice the clock error is often modeled as* $t = (1 + \delta)t^* + \xi(t^*)$*.*

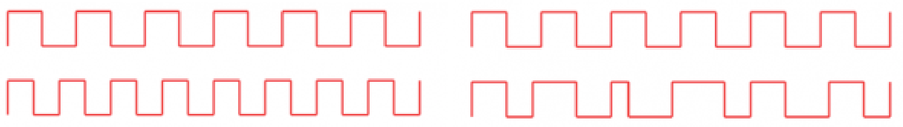


Figure 7.8: Drift (left) and Jitter (right). On top is a square wave, the wall-clock time t^*.

Remarks:

- The importance of accurate timekeeping and clock synchronization is reflected in the following statement by physicist Steven Jefferts: “We’ve learned that every time we build a better clock, somebody comes up with a use for it that you couldn’t have foreseen.”

Definition 7.5 (Drift)**.** *The **drift** δ is the predictable clock error.*

Remarks:

- Drift is relatively constant over time, but may change with supply voltage, temperature and age of an oscillator.
- Stable clock sources, which offer a low drift, are generally preferred, but also more expensive, larger and more power hungry, which is why many consumer products feature inaccurate clocks.

Definition 7.6 (Parts Per Million)**.** *Clock drift is indicated in **parts per million (ppm)**. One ppm corresponds to a time error growth of one microsecond per second.*

Remarks:

- In PCs, the so-called *real-time clock* normally is a crystal oscillator with a maximum drift between 5 and 100 ppm.
- Applications in signal processing, for instance GPS, need more accurate clocks. Common drift values are 0.5 to 2 ppm.

Definition 7.7 (Jitter)**.** *The **jitter** ξ is the unpredictable, random noise of the clock error.*

Remarks:

- In other words, jitter is the irregularity of the clock. Unlike drift, jitter can vary fast.
- Jitter captures all the errors that are not explained by drift. Figure 7.8 visualizes the concepts.

7.2 Clock Synchronization

Definition 7.9 (Clock Synchronization)**.** ***Clock synchronization*** *is the process of matching multiple clocks (nodes) to have a common time.*

Remarks:

- A trade-off exists between synchronization accuracy, convergence time, and cost.
- Different clock synchronization variants may tolerate crashing, erroneous or byzantine nodes.

Algorithm 7.10 Network Time Protocol NTP

```
Two nodes, client $u$ and server $v$

while true do
    Node $u$ sends request to $v$ at time $t_u$
    Node $v$ receives request at time $t_v$
    Node $v$ processes the request and replies at time $t'_v$
    Node $u$ receives the response at time $t'_u$

    Propagation delay $\delta = \frac{(t'_u - t_u) - (t'_v - t_v)}{2}$ (assumption: symmetric)
    Clock skew $\theta = \frac{(t_v - (t_u + \delta)) - (t'_u - (t'_v + \delta))}{2} = \frac{(t_v - t_u) + (t'_v - t'_u)}{2}$
    Node $u$ adjusts clock by $+\theta$
    Sleep before next synchronization
end while
```

Remarks:

- Many NTP servers are public, answering to UDP packets.
- The most accurate NTP servers derive their time from atomic clocks, synchronized to UTC. To reduce those server's load, a hierarchy of NTP servers is available in a forest (multiple trees) structure.
- The regular synchronization of NTP limits the maximum error despite unpredictable clock errors. Synchronizing clocks just once is only sufficient for a short time period.

Definition 7.11 (PTP)**.** *The* ***Precision Time Protocol (PTP)*** *is a clock synchronization protocol similar to NTP, but which uses* ***medium access control (MAC)*** *layer timestamps.*

Remarks:

- MAC layer timestamping removes the unknown time delay incurred through messages passing through the software stack.
- PTP can achieve sub-microsecond accuracy in local networks.

Definition 7.12 (Global Synchronization)**.** ***Global*** *synchronization establishes a common time between* ***any*** *two nodes in the system.*

Remarks:

- For example, email needs global timestamps. Also, event detection for power grid control and earthquake localization need global timestamps.
- Earthquake localization does not need real-time synchronization; it is sufficient if a common time can be reconstructed when needed, also known as “post factum” synchronization.
- NTP and PTP are both examples of clock synchronization algorithms that optimize for global synchronization.
- However, two nodes that constantly communicate may receive their timestamps through different paths of the NTP forest, and hence they may accumulate different errors. Because of the clock skew, a message sent by node u might arrive at node v with a timestamp in the future.

Algorithm 7.13 Local Time Synchronization

```
while true do
    Exchange current time with neighbors
    Adapt time to neighbors, e.g., to average or median
    Sleep before next synchronization
end while
```

Remarks:

- Local synchronization is the method of choice to establish *time-division multiple access (TDMA)* and coordination of wake-up and sleeping times in wireless networks. Only close-by nodes matter as far-away nodes will not interfere with their transmissions.
- Local synchronization is also relevant for precise event localization. For instance, using the speed of sound, measured sound arrival times from co-located sensors can be used to localize a shooter.
- While global synchronization algorithm such as NTP usually synchronize to an external time standard, local algorithms often just synchronize among themselves, i.e., the notion of time does not reflect any time standards.
- In wireless networks, one can simplify and improve synchronization.

Algorithm 7.14 Wireless Clock Synchronization with Known Delays

```
Given: transmitter $s$, receivers $u, v$, with known transmission delays $d_u, d_v$ from transmitter $s$, respectively.

$s$ sends signal at time $t_s$
$u$ receives signal at time $t_u$
$v$ receives signal at time $t_v$

$\Delta_u = t_u - (t_s + d_u)$
$\Delta_v = t_v - (t_s + d_v)$

Clock skew between $u$ and $v$: $\theta = \Delta_v - \Delta_u = t_v - d_v + d_u - t_u$
```

7.3 Time Standards

Definition 7.15 (TAI). *The **International Atomic Time (TAI)** is a time standard derived from over 400 atomic clocks distributed worldwide.*

Remarks:

- Using a weighted average of all involved clocks, TAI is an order of magnitude more stable than the best clock.
- The involved clocks are synchronized using simultaneous observations of GPS or geostationary satellite transmissions using Algorithm 7.14.
- While a single satellite measurement has a time uncertainty on the order of nanoseconds, averaging over a month improves the accuracy by several orders of magnitude.

Definition 7.16 (Leap Second). *A leap second is an extra second added to a minute to make it irregularly 61 instead of 60 seconds long.*

Remarks:

- Time standards use leap seconds to compensate for the slowing of the Earth's rotation. In theory, also negative leap seconds can be used to make some minutes only 59 seconds long. But so far, this was never necessary.
- For easy implementation, not all time standards use leap seconds, for instance TAI and GPS time do not.

Definition 7.17 (UTC). *The **Coordinated Universal Time (UTC)** is a time standard based on TAI with leap seconds added at irregular intervals to keep it close to mean solar time at* 0° *longitude.*

Remarks:

- The global time standard *Greenwich Mean Time (GMT)* was already established in 1884. With the invention of caesium atomic clocks and the subsequent redefinition of the SI second, UTC replaced GMT in 1967.
- Before time standards existed, each city set their own time according to the local mean solar time, which is difficult to measure exactly. This was changed by the upcoming rail and communication networks.
- Different notations for time and date are in use. A standardized format for timestamps, mostly used for processing by computers, is the ISO 8601 standard. According to this standard, a UTC timestamp looks like this: `1712-02-30T07:39:52Z`. `T` separates the date and time parts while `Z` indicates the time zone with zero offset from UTC.
- Why UTC and not "CUT"? Because France insisted. Same for other abbreviations in this domain, e.g. TAI.

Definition 7.18 (Time Zone)**.** *A **time zone** is a geographical region in which the same time offset from UTC is officially used.*

Remarks:

- Time zones serve to roughly synchronize noon with the sun reaching the day's highest apparent elevation angle.
- Some time zones' offset is not a whole number of hours, e.g. India.

7.4 Clock Sources

Definition 7.19 (Atomic Clock)**.** *An **atomic clock** is a clock which keeps time by counting oscillations of atoms.*

Remarks:

- Atomic clocks are the most accurate clocks known. They can have a drift of only about one second in 150 million years, about 2e-10 ppm!
- Many atomic clocks are based on caesium atoms, which led to the current definition of a second. Others use hydrogen-1 or rubidium-87.
- In the future, atoms with higher frequency oscillations could yield even more accurate clocks.
- Atomic clocks are getting smaller and more energy efficient. Chip-scale atomic clocks (CSAC) are currently being produced for space applications and may eventually find their way into consumer electronics.

Definition 7.20 (System Clock)**.** *The **system clock** in a computer is an oscillator used to synchronize all components on the motherboard.*

Remarks:

- Usually, a quartz crystal oscillator with a frequency of some tens to hundreds MHz is used.
- Therefore, the system clock can achieve a precision of some ns!
- The *CPU clock* is usually a multiple of the system clock, generated from the system clock through a clock multiplier.
- To guarantee nominal operation of the computer, the system clock must have low jitter. Otherwise, some components might not get enough time to complete their operation before the next (early) clock pulse arrives.
- Drift however is not critical for system stability.
- Applications of the system clock include thread scheduling and ensuring smooth media playback.
- If a computer is shut down, the system clock is not running; it is reinitialized when starting the computer.

Definition 7.21 (RTC)**.** *The **real-time clock (RTC)** in a computer is a battery backed oscillator which is running even if the computer is shut down or unplugged.*

Remarks:

- The RTC is read at system startup to initialize the system clock.
- This keeps the computer's time close to UTC even when the time cannot be synchronized over a network.
- RTCs are relatively inaccurate, with a common maximum drift of 5, 20 or even 100 ppm, depending on quality and temperature.
- In many cases, the RTC frequency is 32.768 kHz, which allows for simple timekeeping based on binary counter circuits because the frequency is exactly 2^{15} Hz.

Definition 7.22 (Radio Time Signal)**.** *A **Radio Time Signal** is a time code transmitted via radio waves by a time signal station, referring to a time in a given standard such as UTC.*

Remarks:

- Time signal stations use atomic clocks to send as accurate time codes as possible.
- Radio-controlled clocks are an example application of radio signal time synchronization.
- In Europe, most radio-controlled clocks use the signal transmitted by the *DCF77* station near Frankfurt, Germany.

- Radio time signals can be received much farther than the horizon of the transmitter due to signal reflections at the ionosphere. DCF77 for instance has an official range of 2,000 km.
- The time synchronization accuracy when using radio time signals is limited by the propagation delay of the signal. For instance the delay Frankfurt-Zurich is about 1 ms.

Definition 7.23 (Power Line Clock)**.** *A **power line clock** measures the oscillations from electric AC power lines, e.g. 50 Hz.*

Remarks:

- Clocks in kitchen ovens are usually driven by power line oscillations.
- AC power line oscillations drift about 10 ppm, which is remarkably stable.
- The magnetic field radiating from power lines is strong enough that power line clocks can work wirelessly.
- Power line clocks can be synchronized by matching the observed noisy power line oscillation patterns.
- Power line clocks operate with as little as a few ten µW.

Definition 7.24 (Sunlight Time Synchronization)**.** ***Sunlight time synchronization** is a method of reconstructing global timestamps by correlating annual solar patterns from light sensors' length of day measurements.*

Remarks:

- Sunlight time synchronization is relatively inaccurate.
- Due to low data rates from length of day measurements, sunlight time synchronization is well-suited for long-time measurements with data storage and post-processing, requiring no communication at the time of measurement.
- Historically, sun and lunar observations were the first measurements used for time determination. Some clock towers still feature sun dials.
- . . . but today, the most popular source of time is probably GPS!

7.5 GPS

Definition 7.25 (Global Positioning System)**.** *The **Global Positioning System (GPS)** is a **Global Navigation Satellite System (GNSS)**, consisting of at least 24 satellites orbiting around the Earth, each continuously transmitting its position and time code.*

Remarks:

- Positioning is done in space and *time*!
- GPS provides position and time information to receivers anywhere on Earth where at least four satellite signals can be received.
- Line of sight (LOS) between satellite and receiver is advantageous. GPS works poorly indoors, or with reflections.
- Besides the US GPS, three other GNSS exist: the European Galileo, the Russian GLONASS and the Chinese BeiDou.
- GPS satellites orbit around Earth approximately 20,000 km above the surface, circling Earth twice a day. The signals take between 64 and 89 ms to reach Earth.
- The orbits are precisely determined by ground control stations, optimized for a high number of satellites being concurrently above the horizon at any place on Earth.

Algorithm 7.26 GPS Satellite

```
Given: Each satellite has a unique 1023 bit (±1, see below) $PRN$ sequence,
  plus some current navigation data $D$ (also ±1).
The code below is a bit simplified, concentrating on the digital aspects,
  ignoring that the data is sent on a carrier frequency of 1575.42 MHz.

while true do
  for all bits $D_i \in D$ do
    for $j = 0 \dots 19$ do
      for $k = 0 \dots 1022$ do {this loop takes exactly 1 ms}
        Send bit $PRN_k \cdot D_i$
      end for
    end for
  end for
end while
```

Definition 7.27 (PRN)**.** ***Pseudo-Random Noise (PRN)*** *sequences are pseudo-random bit strings. Each GPS satellite uses a unique PRN sequence with a length of 1023 bits for its signal transmissions.*

Remarks:

- The GPS PRN sequences are so-called *Gold codes*, which have low cross-correlation with each other.
- To simplify our math (abstract from modulation), each PRN bit is either 1 or -1.

Definition 7.28 (Navigation Data)**.** ***Navigation Data*** *is the data transmitted from satellites, which includes orbit parameters to determine satellite positions, timestamps of signal transmission, atmospheric delay estimations and status information of the satellites and GPS as a whole, such as the accuracy and validity of the data.*

Remarks:

- As seen in Algorithm 7.26 each bit is repeated 20 times for better robustness. Thus, the navigation data rate is only 50 bit/s.
- Due to this limited data rate, timestamps are sent every 6 seconds, satellite orbit parameters (function of the satellite position over time) only every 30 seconds. As a result, the latency of a first position estimate after turning on a receiver, which is called *time-to-first-fix (TTFF)*, can be high.

Definition 7.29 (Circular Cross-Correlation)**.** *The **circular cross-correlation** is a similarity measure between two vectors of length N, **circularly** shifted by a given displacement d:*

$$cxcorr(\boldsymbol{a}, \boldsymbol{b}, d) = \sum_{i=0}^{N-1} a_i \cdot b_{i+d \bmod N}$$

Remarks:

- The two vectors are most similar at the displacement d where the sum (cross-correlation value) is maximum.
- The vector of cross-correlation values with all N displacements can efficiently be computed using a fast Fourier transform (FFT) in $\mathcal{O}(N \log N)$ instead of $\mathcal{O}(N^2)$ time.

Algorithm 7.30 Acquisition

```
Received 1 ms signal $\boldsymbol{s}$ with sampling rate $r \cdot 1,023$ kHz
Possible Doppler shifts $F$, e.g. {-10 kHz, -9.8 kHz, ..., +10 kHz}
Tensor $A = 0$: Satellite × carrier frequency × time

for all satellites $i$ do
  $PRN'_i = PRN_i$ stretched with ratio $r$
  for all Doppler shifts $f \in F$ do
    Build modulated $PRN''_i$ with $PRN'_i$ and Doppler frequency $f$
    for all delays $d \in \{0, 1, \dots, 1,023 \cdot r - 1\}$ do
      $A_i(f, d) = |cxcorr(\boldsymbol{s}, \boldsymbol{PRN''_i}, d)|$
    end for
  end for
  Select $d^*$ that maximizes $\max_d \max_f A_i(f, d)$
  Signal arrival time $r_i = d^*/(r \cdot 1,023 \text{ kHz})$
end for
```

Remarks:

- Multiple milliseconds of acquisition can be summed up to average out noise and therefore improve the arrival time detection probability.

Definition 7.31 (Acquisition)**.** ***Acquisition** is the process in a GPS receiver that finds the visible satellite signals and detects the delays of the PRN sequences and the Doppler shifts of the signals.*

Remarks:

- The relative speed between satellite and receiver introduces a significant Doppler shift to the carrier frequency. In order to decode the signal, a frequency search for the Doppler shift is necessary.

- The nested loops make acquisition the computationally most intensive part of a GPS receiver.

Algorithm 7.32 Classic GPS Receiver

```
$h$: Unknown receiver *handset* position
$\theta$: Unknown handset time offset to GPS system time
$r_i$: measured signal arrival time in *handset time system*
$c$: signal propagation speed (GPS: speed of light)

Perform Acquisition (Algorithm 7.30)
Track signals and decode navigation data
for all satellites $i$ do
    Using navigation data, determine signal transmit time $s_i$ and position $p_i$
    Measured satellite transmission delay $d_i = r_i - s_i$
end for
Solve the following system of equations for $h$ and $\theta$:
$||p_i - h||/c = d_i - \theta$, for all $i$
```

Remarks:

- GPS satellites carry precise atomic clocks, but the receiver is not synchronized with the satellites. The arrival times of the signals at the receiver are determined in the receiver's local time. Therefore, even though the satellite signals include transmit timestamps, the exact distance between satellites and receiver is unknown.

- In total, the positioning problem contains four unknown variables, three for the handset's spatial position and one for its time offset from the system time. Therefore, signals from at least four transmitters are needed to find the correct solution.

- Since the equations are quadratic (distance), with as many observations as variables, the system of equations has two solutions in principle. For GPS however, in practice one of the solutions is far from the Earth surface, so the correct solution can always be identified without a fifth satellite.

- More received signals help reducing the measurement noise and thus improving the accuracy.

- Since the positioning solution, which is also called position fix, includes the handset's time offset Δ, this establishes a global time for all handsets. Thus, GPS is useful for global time synchronization.

- For a handset with unknown position, GPS timing is more accurate than time synchronization with a single transmitter, like a time signal station (cf. Definition 7.22). With the latter, the unknown signal propagation delays cannot be accounted for.

Definition 7.33 (A-GPS)**.** *An **Assisted GPS (A-GPS)** receiver fetches the satellite orbit parameters and other navigation data from the Internet, for instance via a cellular network.*

Remarks:

- A-GPS reduces the data transmission time, and thus the TTFF, from a maximum of 30 seconds per satellite to a maximum of 6 seconds.
- Smartphones regularly use A-GPS. However, coarse positioning is usually done based on nearby Wi-Fi base stations only, which saves energy compared to GPS.
- Another GPS improvement is *Differential GPS (DGPS)*: A receiver with a fixed location within a few kilometers of a mobile receiver compares the observed and actual satellite distances. This error is then subtracted at the mobile receiver. DGPS achieves accuracies in the order of 10 cm.

Definition 7.34 (Snapshot GPS Receiver)**.** *A **snapshot receiver** is a GPS receiver that captures one or a few milliseconds of raw GPS signal for a position fix.*

Remarks:

- Snapshot receivers aim at the remaining latency that results from the transmission of timestamps from the satellites every six seconds.
- Since time changes continuously, timestamps cannot be fetched together with the satellite orbit parameters that are valid for two hours.
- A snapshot receiver can determine the ranges to the satellites modulo 1 ms, which corresponds to 300 km. An approximate time and location of the receiver is used to resolve these ambiguities without a timestamp from the satellite signals themselves.

Definition 7.35 (CTN)**.** ***Coarse Time Navigation (CTN)** is a snapshot receiver positioning technique measuring sub-millisecond satellite ranges from correlation peaks, like conventional GPS receivers.*

Remarks:

- A CTN receiver determines the signal transmit times and satellite positions from its own approximate location by subtracting the signal propagation delay from the receive time. The receiver location and time is not exactly known, but since signals are transmitted exactly at whole milliseconds, rounding to the nearest whole millisecond gives the signal transmit time.

- With only a few milliseconds of signal, noise cannot be averaged out well and may lead to wrong signal arrival time estimates. Such wrong measurements usually render the system of equations unsolvable, making positioning infeasible.

Algorithm 7.36 Collective Detection Receiver

```
Given: A raw 1 ms GPS sample $\boldsymbol{s}$, a set $H$ of location/time hypotheses
In addition, the receiver learned all navigation and atmospheric data

for all hypotheses $h \in H$ do
    Vector $\boldsymbol{r} = 0$
    Set $V$ = satellites that should be visible with hypothesis $h$
    for all satellites $i$ in $V$ do
        $\boldsymbol{r} = \boldsymbol{r} + \boldsymbol{r_i}$, where $\boldsymbol{r_i}$ is expected signal of satellite $i$. The data of vector $\boldsymbol{r_i}$ incorporates all available information: distance and atmospheric delay between satellite and receiver, frequency shift because of Doppler shift due to satellite movement, current navigation data bit of satellite, etc.
    end for
    Probability $P_h = cxcorr(\boldsymbol{s}, \boldsymbol{r}, 0)$
end for
Solution: hypothesis $h \in H$ maximizing $P_h$
```

Definition 7.37 (Collective Detection)**.** ***Collective detection (CD)*** *is a maximum likelihood snapshot receiver localization method, which does not determine an arrival time for each satellite, but rather combine all the available information and take a decision only at the end of the computation.*

Remarks:

- CD can tolerate a few low quality satellite signals and is thus more robust than CTN.
- In essence, CD tests how well position hypotheses match the received signal. For large position and time uncertainties, the high number of hypotheses require a lot of computation power.
- CD can be sped up by a branch and bound approach, which reduces the computation per position fix to the order of one second even for uncertainties of 100 km and a minute.

7.6 Lower Bounds

In the *clock synchronization* problem, we are given a network (graph) with n nodes. The goal for each node is to have a (logical) clock such that the clock values are well synchronized, and close to real time. Each node is equipped with a hardware (system) clock, that ticks more or less in real time, i.e., the time between two pulses is arbitrary between $[1-\epsilon, 1+\epsilon]$, for a constant $\epsilon \ll 1$. We assume that messages sent over the edges of the graph have a delivery time between $[0, 1]$. In other words, we have a bounded but variable drift on the hardware clocks and an arbitrary jitter in the delivery times. The goal is to design a message-passing algorithm that ensures that the logical clock skew of adjacent nodes is as small as possible at all times.

Definition 7.38 (Local and Global Clock Skew)**.** *In a network of nodes, the* ***local clock skew*** *is the skew between neighboring nodes, while the* ***global clock skew*** *is the maximum skew between any two nodes.*

Remarks:

- Of interest is also the *average global clock skew*, that is the average skew between any pair of nodes.

Theorem 7.39. *The global clock skew (Definition 7.12) is $\Omega(D)$, where D is the diameter of the network graph.*

Proof. For a node u, let t_u be the logical time of u and let $(u \to v)$ denote a message sent from u to a node v. Let $t(m)$ be the time delay of a message m and let u and v be neighboring nodes. First consider a case where the message delays between u and v are $1/2$. Then, all the messages sent by u and v at time t according to the clock of the sender arrive at time $t + 1/2$ according to the clock of the receiver.

Then consider the following cases

- $t_u = t_v + 1/2$, $t(u \to v) = 1$, $t(v \to u) = 0$
- $t_u = t_v - 1/2$, $t(u \to v) = 0$, $t(v \to u) = 1$,

where the message delivery time is always fast for one node and slow for the other and the logical clocks are off by $1/2$. In both scenarios, the messages sent at time i according to the clock of the sender arrive at time $i + 1/2$ according to the logical clock of the receiver. Therefore, for nodes u and v, both cases with clock drift seem the same as the case with perfectly synchronized clocks. Furthermore, in a linked list of D nodes, the left- and rightmost nodes l, r cannot distinguish $t_l = t_r + D/2$ from $t_l = t_r - D/2$. □

Remarks:

- From Theorem 7.39, it directly follows that any reasonable clock synchronization algorithm must have a global skew of $\Omega(D)$.
- Many natural algorithms manage to achieve a global clock skew of $\mathcal{O}(D)$.
- As both message jitter and hardware clock drift are bounded by constants, it feels like we should be able to get a constant drift at least between neighboring nodes.
- Let us look at the following algorithm:

Lemma 7.41. *The clock synchronization protocol of Algorithm 7.40 has a local skew of $\Omega(n)$.*

Algorithm 7.40 Local Clock Synchronization (at node v)

```
repeat
  send logical time $t_v$ to all neighbors
  if Receive logical time $t_u$, where $t_u > t_v$, from any neighbor $u$ then
    $t_v = t_u$
  end if
until done
```

Proof. Let the graph be a linked list of D nodes. We denote the nodes by $v_1, v_2, \ldots, v_D$ from left to right and the logical clock of node v_i by t_i. Apart from the left-most node v_1 all hardware clocks run with speed 1 (real time). Node v_1 runs at maximum speed, i.e. the time between two pulses is not 1 but $1 - \epsilon$. Assume that initially all message delays are 1. After some time, node v_1 will start to speed up v_2, and after some more time v_2 will speed up v_3, and so on. At some point of time, we will have a clock skew of 1 between any two neighbors. In particular $t_1 = t_D + D - 1$.

Now we start playing around with the message delays. Let $t_1 = T$. First we set the delay between the v_1 and v_2 to 0. Now node v_2 immediately adjusts its logical clock to T. After this event (which is instantaneous in our model) we set the delay between v_2 and v_3 to 0, which results in v_3 setting its logical clock to T as well. We perform this successively to all pairs of nodes until v_{D-2} and v_{D-1}. Now node v_{D-1} sets its logical clock to T, which indicates that the difference between the logical clocks of v_{D-1} and v_D is $T - (T - (D - 1)) = D - 1$. □

Remarks:

- The introduced examples may seem cooked-up, but examples like this exist in all networks, and for all algorithms. Indeed, it was shown that any natural clock synchronization algorithm must have a bad local skew. In particular, a protocol that averages between all neighbors (like Algorithm 7.13) is even worse than Algorithm 7.40. An averaging algorithm has a clock skew of $\Omega(D^2)$ in the linked list, at all times.
- It was shown that the local clock skew is $\Theta(\log D)$, i.e., there is a protocol that achieves this bound, and there is a proof that no algorithm can be better than this bound!
- Note that these are worst-case bounds. In practice, clock drift and message delays may not be the worst possible, typically the speed of hardware clocks changes at a comparatively slow pace and the message transmission times follow a benign probability distribution. If we assume this, better protocols do exist, in theory as well as in practice.

Chapter Notes

Atomic clocks can be used as a GPS fallback for data center synchronization [CDE+13].

GPS has been such a technological breakthrough that even though it dates back to the 1970s, the new GNSS still use essentially the same techniques. Several people worked on snapshot GPS receivers, but the technique has not penetrated into commercial receivers yet. Liu et al. [LPH+12] presented a practical CTN receiver and reduced the solution space by eliminating solutions not lying on the ground. CD receivers are studied since at least 2011 [ABD+11] and have recently been made practically feasible through branch and bound [BEW17]

It has been known for a long time that the global clock skew is $\Theta(D)$ [LL84, ST87]. The problem of synchronizing the clocks of nearby nodes was introduced by Fan and Lynch in [LF04]; they proved a surprising lower bound of $\Omega(\log D / \log\log D)$ for the local skew. The first algorithm providing a non-trivial local skew of $\mathcal{O}(\sqrt{D})$ was given in [LW06]. Later, matching upper and lower bounds of $\Theta(\log D)$ were given in [LLW10]. The problem has also been studied in a dynamic setting [KLO09, KLLO10] or when a fraction of nodes experience byzantine faults and the other nodes have to recover from faulty initial state (i.e., self-stabilizing) [DD06, DW04]. The self-stabilizing byzantine case has been solved with asymptotically optimal skew [KL18].

Clock synchronization is a well-studied problem in practice, for instance regarding the global clock skew in sensor networks, e.g. [EGE02, GKS03, MKSL04, PSJ04]. One more recent line of work is focussing on the problem of minimizing the local clock skew [BvRW07, SW09, LSW09, FW10, FZTS11].

This chapter was written in collaboration with Manuel Eichelberger.

Bibliography

[ABD+11] Penina Axelrad, Ben K Bradley, James Donna, Megan Mitchell, and Shan Mohiuddin. Collective Detection and Direct Positioning Using Multiple GNSS Satellites. *Navigation*, 58(4):305–321, 2011.

[BEW17] Pascal Bissig, Manuel Eichelberger, and Roger Wattenhofer. Fast and Robust GPS Fix Using One Millisecond of Data. In *Information Processing in Sensor Networks (IPSN), 2017 16th ACM/IEEE International Conference on*, pages 223–234. IEEE, 2017.

[BvRW07] Nicolas Burri, Pascal von Rickenbach, and Roger Wattenhofer. Dozer: Ultra-Low Power Data Gathering in Sensor Networks. In *International Conference on Information Processing in Sensor Networks (IPSN), Cambridge, Massachusetts, USA*, April 2007.

[CDE+13] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, Jeffrey John Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[DD06] Ariel Daliot and Danny Dolev. Self-Stabilizing Byzantine Pulse Synchronization. *Computing Research Repository*, 2006.

[DW04] Shlomi Dolev and Jennifer L. Welch. Self-stabilizing clock synchronization in the presence of Byzantine faults. September 2004.

[EGE02] Jeremy Elson, Lewis Girod, and Deborah Estrin. Fine-grained Network Time Synchronization Using Reference Broadcasts. *ACM SIGOPS Operating Systems Review*, 36:147–163, 2002.

[FW10] Roland Flury and Roger Wattenhofer. Slotted Programming for Sensor Networks. In *International Conference on Information Processing in Sensor Networks (IPSN), Stockholm, Sweden*, April 2010.

[FZTS11] Federico Ferrari, Marco Zimmerling, Lothar Thiele, and Olga Saukh. Efficient Network Flooding and Time Synchronization with Glossy. In *Proceedings of the 10th International Conference on Information Processing in Sensor Networks (IPSN)*, pages 73–84, 2011.

[GKS03] Saurabh Ganeriwal, Ram Kumar, and Mani B. Srivastava. Timing-sync Protocol for Sensor Networks. In *Proceedings of the 1st international conference on Embedded Networked Sensor Systems (SenSys)*, 2003.

[KL18] Pankaj Khanchandani and Christoph Lenzen. Self-Stabilizing Byzantine Clock Synchronization with Optimal Precision. January 2018.

[KLLO10] Fabian Kuhn, Christoph Lenzen, Thomas Locher, and Rotem Oshman. Optimal Gradient Clock Synchronization in Dynamic Networks. In *29th Symposium on Principles of Distributed Computing (PODC), Zurich, Switzerland*, July 2010.

[KLO09] Fabian Kuhn, Thomas Locher, and Rotem Oshman. Gradient Clock Synchronization in Dynamic Networks. In *21st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), Calgary, Canada*, August 2009.

[LF04] Nancy Lynch and Rui Fan. Gradient Clock Synchronization. In *Proceedings of the 23rd Annual ACM Symposium on Principles of Distributed Computing (PODC)*, 2004.

[LL84] Jennifer Lundelius and Nancy Lynch. An Upper and Lower Bound for Clock Synchronization. *Information and Control*, 62:190–204, 1984.

[LLW10] Christoph Lenzen, Thomas Locher, and Roger Wattenhofer. Tight Bounds for Clock Synchronization. In *Journal of the ACM, Volume 57, Number 2*, January 2010.

[LPH$^+$12] Jie Liu, Bodhi Priyantha, Ted Hart, Heitor Ramos, Antonio A.F. Loureiro, and Qiang Wang. Energy Efficient GPS Sensing with Cloud Offloading. In *10th ACM Conference on Embedded Networked Sensor Systems (SenSys 2012)*. ACM, November 2012.

[LSW09] Christoph Lenzen, Philipp Sommer, and Roger Wattenhofer. Optimal Clock Synchronization in Networks. In *7th ACM Conference on Embedded Networked Sensor Systems (SenSys), Berkeley, California, USA*, November 2009.

[LW06] Thomas Locher and Roger Wattenhofer. Oblivious Gradient Clock Synchronization. In *20th International Symposium on Distributed Computing (DISC), Stockholm, Sweden*, September 2006.

[MKSL04] Miklós Maróti, Branislav Kusy, Gyula Simon, and Ákos Lédeczi. The Flooding Time Synchronization Protocol. In *Proceedings of the 2nd international Conference on Embedded Networked Sensor Systems*, SenSys '04, 2004.

[PSJ04] Santashil PalChaudhuri, Amit Kumar Saha, and David B. Johnson. Adaptive Clock Synchronization in Sensor Networks. In *Proceedings of the 3rd International Symposium on Information Processing in Sensor Networks*, IPSN '04, 2004.

[ST87] T. K. Srikanth and S. Toueg. Optimal Clock Synchronization. *Journal of the ACM*, 34:626–645, 1987.

[SW09] Philipp Sommer and Roger Wattenhofer. Gradient Clock Synchronization in Wireless Sensor Networks. In *8th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN), San Francisco, USA*, April 2009.

Chapter 8

Quorum Systems

What happens if a single server is no longer powerful enough to service all your customers? The obvious choice is to add more servers and to use the majority approach (e.g. Paxos, Chapter 2) to guarantee consistency. However, even if you buy one million servers, a client still has to access more than half of them per request! While you gain fault-tolerance, your efficiency can at most be doubled. Do we have to give up on consistency?

Let us take a step back: We used majorities because majority sets always overlap. But are majority sets the only sets that guarantee overlap? In this chapter we study the theory behind overlapping sets, known as quorum systems.

Definition 8.1 (quorum, quorum system)**.** *Let $V = \{v_1, \ldots, v_n\}$ be a set of nodes. A* ***quorum*** *$Q \subseteq V$ is a subset of these nodes. A* ***quorum system*** *$\mathcal{S} \subset 2^V$ is a set of quorums s.t. every two quorums intersect, i.e., $Q_1 \cap Q_2 \neq \emptyset$ for all $Q_1, Q_2 \in \mathcal{S}$.*

Remarks:

- When a quorum system is being used, a client selects a quorum, acquires a lock (or ticket) on all nodes of the quorum, and when done releases all locks again. The idea is that no matter which quorum is chosen, its nodes will intersect with the nodes of every other quorum.
- What can happen if two quorums try to lock their nodes at the same time?
- A quorum system $\mathcal{S}$ is called **minimal** if $\forall Q_1, Q_2 \in \mathcal{S} : Q_1 \not\subset Q_2$.
- The simplest quorum system imaginable consists of just one quorum, which in turn just consists of one server. It is known as **Singleton**.
- In the **Majority** quorum system, every quorum has $\lfloor \frac{n}{2} \rfloor + 1$ nodes.
- Can you think of other simple quorum systems?

8.1 Load and Work

Definition 8.2 (access strategy)**.** *An* ***access strategy*** *Z defines the probability $P_Z(Q)$ of accessing a quorum $Q \in \mathcal{S}$ s.t. $\sum_{Q \in \mathcal{S}} P_Z(Q) = 1$.*

Definition 8.3 (load)**.**

- *The* ***load*** *of access strategy Z on a node v_i is $L_Z(v_i) = \sum_{Q \in \mathcal{S}; v_i \in Q} P_Z(Q)$.*
- *The* ***load*** *induced by access strategy Z on a quorum system $\mathcal{S}$ is the maximal load induced by Z on any node in $\mathcal{S}$, i.e., $L_Z(\mathcal{S}) = \max_{v_i \in \mathcal{S}} L_Z(v_i)$.*
- *The* ***load*** *of a quorum system $\mathcal{S}$ is $L(\mathcal{S}) = \min_Z L_Z(\mathcal{S})$.*

Definition 8.4 (work)**.**

- *The* ***work*** *of a quorum $Q \in \mathcal{S}$ is the number of nodes in Q, $W(Q) = |Q|$.*
- *The* ***work*** *induced by access strategy Z on a quorum system $\mathcal{S}$ is the expected number of nodes accessed, i.e., $W_Z(\mathcal{S}) = \sum_{Q \in \mathcal{S}} P_Z(Q) \cdot W(Q)$.*
- *The* ***work*** *of a quorum system $\mathcal{S}$ is $W(\mathcal{S}) = \min_Z W_Z(\mathcal{S})$.*

Remarks:

- Note that you cannot choose different access strategies Z for work and load, you have to pick a single Z for both.
- We illustrate the above concepts with a small example. Let $V = \{v_1, v_2, v_3, v_4, v_5\}$ and $\mathcal{S} = \{Q_1, Q_2, Q_3, Q_4\}$, with $Q_1 = \{v_1, v_2\}$, $Q_2 = \{v_1, v_3, v_4\}$, $Q_3 = \{v_2, v_3, v_5\}$, $Q_4 = \{v_2, v_4, v_5\}$. If we choose the access strategy Z s.t. $P_Z(Q_1) = 1/2$ and $P_Z(Q_2) = P_Z(Q_3) = P_Z(Q_4) = 1/6$, then the node with the highest load is v_2 with $L_Z(v_2)$ $= 1/2 + 1/6 + 1/6 = 5/6$, i.e., $L_Z(\mathcal{S}) = 5/6$. Regarding work, we have $W_Z(\mathcal{S}) = 1/2 \cdot 2 + 1/6 \cdot 3 + 1/6 \cdot 3 + 1/6 \cdot 3 = 15/6$.
- Can you come up with a better access strategy for $\mathcal{S}$?
- If every quorum Q in a quorum system $\mathcal{S}$ has the same number of elements, $\mathcal{S}$ is called *uniform.*
- What is the minimum load a quorum system can have?

Primary Copy vs. Majority		**Singleton**	**Majority**
How many nodes need to be accessed?	(Work)	**1**	$> n/2$
What is the load of the busiest node?	(Load)	1	$> \mathbf{1/2}$

Table 8.5: First comparison of the Singleton and Majority quorum systems. Note that the Singleton quorum system can be a good choice when the failure probability of every single node is > 1/2.

Theorem 8.6. *Let $\mathcal{S}$ be a quorum system. Then $L(\mathcal{S}) \geq 1/\sqrt{n}$ holds.*

Proof. Let $Q = \{v_1, \ldots, v_q\}$ be a quorum of minimal size in $\mathcal{S}$, with $|Q| = q$. Let Z be an access strategy for $\mathcal{S}$. Every other quorum in $\mathcal{S}$ intersects in at least one element with this quorum Q. Each time a quorum is accessed, at least one node in Q is accessed as well, yielding a lower bound of $L_Z(v_i) \geq 1/q$ for some $v_i \in Q$.

Furthermore, as Q is minimal, at least q nodes need to be accessed, yielding $W(\mathcal{S}) \geq q$. Thus, $L_Z(v_i) \geq q/n$ for some $v_i \in Q$, as each time q nodes are accessed, the load of the most accessed node is at least q/n.

Combining both ideas leads to $L_Z(\mathcal{S}) \geq \max(1/q, q/n) \Rightarrow L_Z(\mathcal{S}) \geq 1/\sqrt{n}$. Thus, $L(\mathcal{S}) \geq 1/\sqrt{n}$, as Z can be *any* access strategy. □

Remarks:

- Can we achieve this load?

8.2 Grid Quorum Systems

Definition 8.7 (Basic Grid quorum system). *Assume $\sqrt{n} \in \mathbb{N}$, and arrange the n nodes in a square matrix with side length of $\sqrt{n}$, i.e., in a grid. The basic **Grid** quorum system consists of $\sqrt{n}$ quorums, with each containing the full row i and the full column i, for $1 \leq i \leq \sqrt{n}$.*

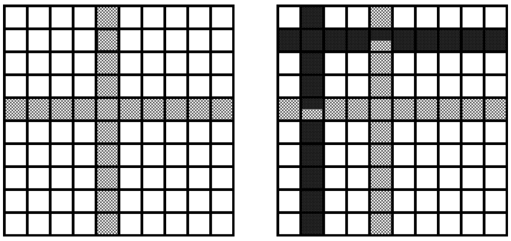


Figure 8.8: The basic version of the Grid quorum system, where each quorum Q_i with $1 \leq i \leq \sqrt{n}$ uses row i and column i. The size of each quorum is $2\sqrt{n} - 1$ and two quorums overlap in exactly two nodes. Thus, when the access strategy Z is uniform (i.e., the probability of each quorum is $1/\sqrt{n}$), the work is $2\sqrt{n} - 1$, and the load of every node is in $\Theta(1/\sqrt{n})$.

Remarks:

- Consider the right picture in Figure 8.8: The two quorums intersect in two nodes. If both quorums were to be accessed at the same time, it is not guaranteed that at least one quorum will lock all of its nodes, as they could enter a deadlock!
- In the case of just two quorums, one could solve this by letting the quorums just intersect in one node, see Figure 8.9. However, already with three quorums the same situation could occur again, progress is not guaranteed!
- However, by deviating from the "access all at once" strategy, we can guarantee progress if the nodes are totally ordered!

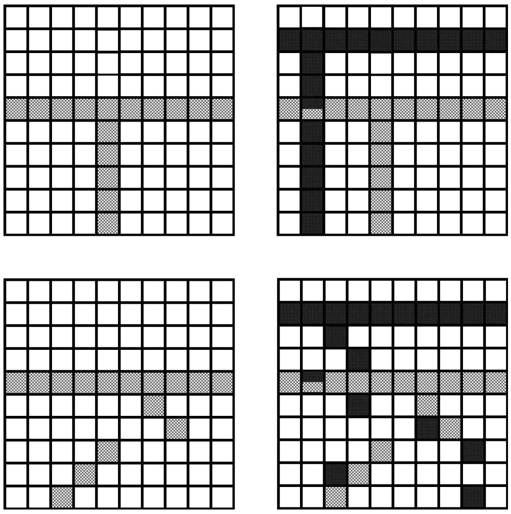


Figure 8.9: There are other ways to choose quorums in the grid s.t. pairwise different quorums only intersect in one node. The size of each quorum is between $\sqrt{n}$ and $2\sqrt{n} - 1$, i.e., the work is in $\Theta(\sqrt{n})$. When the access strategy Z is uniform, the load of every node is in $\Theta(1/\sqrt{n})$.

Algorithm 8.10 Sequential Locking Strategy for a Quorum Q

```
Attempt to lock the nodes one by one, ordered by their identifiers
Should a node be already locked, release all locks and start over
```

Theorem 8.11. *If each quorum is accessed by Algorithm 8.10, at least one quorum will obtain a lock for all of its nodes.*

Proof. We prove the theorem by contradiction. Assume no quorum can make progress, i.e., for every quorum we have: At least one of its nodes is locked by another quorum. Let v be the node with the highest identifier that is locked by some quorum Q. Observe that Q already locked all of its nodes with a smaller identifier than v, otherwise Q would have restarted. As all nodes with a higher identifier than v are not locked, Q either has locked all of its nodes or can make progress – a contradiction. As the set of nodes is finite, one quorum will eventually be able to lock all of its nodes. □

Remarks:

- But now we are back to sequential accesses in a distributed system? Let's do it concurrently with the same idea, i.e., resolving conflicts by the ordering of the nodes. Then, a quorum that locked the highest identifier so far can always make progress!

Theorem 8.13. *If the nodes and quorums use Algorithm 8.12, at least one quorum will obtain a lock for all of its nodes.*

Algorithm 8.12 Concurrent Locking Strategy for a Quorum Q

Invariant: Let $v_Q \in Q$ be the highest identifier of a node locked by Q s.t. all nodes $v_i \in Q$ with $v_i < v_Q$ are locked by Q as well. Should Q not have any lock, then v_Q is set to 0.

```
repeat
  Attempt to lock all nodes of the quorum $Q$
  for each node $v \in Q$ that was not able to be locked by $Q$ do
    exchange $v_Q$ and $v_{Q'}$ with the quorum $Q'$ that locked $v$
    if $v_Q > v_{Q'}$ then
      $Q'$ releases lock on $v$ and $Q$ acquires lock on $v$
    end if
  end for
until all nodes of the quorum $Q$ are locked
```

Proof. The proof is analogous to the proof of Theorem 8.11: Assume for contradiction that no quorum can make progress. However, at least the quorum with the highest v_Q can always make progress – a contradiction! As the set of nodes is finite, at least one quorum will eventually be able to acquire a lock on all of its nodes. □

Remarks:

- What if a quorum locks all of its nodes and then crashes? Is the quorum system dead now? This issue can be prevented by, e.g., using leases instead of locks: leases have a timeout, i.e., a lock is released eventually.

8.3 Fault Tolerance

Definition 8.14 (resilience)**.** *If any f nodes from a quorum system $\mathcal{S}$ can fail s.t. there is still a quorum $Q \in \mathcal{S}$ without failed nodes, then $\mathcal{S}$ is f-**resilient**. The largest such f is the **resilience** $R(\mathcal{S})$.*

Theorem 8.15. *Let $\mathcal{S}$ be a Grid quorum system where each of the n quorums consists of a full row and a full column. $\mathcal{S}$ has a resilience of $\sqrt{n} - 1$.*

Proof. If all $\sqrt{n}$ nodes on the diagonal of the grid fail, then every quorum will have at least one failed node. Should less than $\sqrt{n}$ nodes fail, then there is a row and a column without failed nodes. □

Remarks:

- The Grid quorum system in Theorem 8.15 is different from the Basic Grid quorum system described in Definition 8.7. In each quorum in the Basic Grid quorum system the row and column index are identical, while in the Grid quorum system of Theorem 8.15 this is not the case.

Definition 8.16 (failure probability)**.** *Assume that every node works with a fixed probability p (in the following we assume concrete values, e.g. $p > 1/2$). The **failure probability** $F_p(\mathcal{S})$ of a quorum system $\mathcal{S}$ is the probability that at least one node of every quorum fails.*

Remarks:

- The **asymptotic failure probability** is $F_p(\mathcal{S})$ for $n \to \infty$.

Facts 8.17. *A version of a* ***Chernoff bound*** *states the following:*
Let $x_1, \ldots, x_n$ be independent Bernoulli-distributed random variables with $Pr[x_i = 1] = p_i$ and $Pr[x_i = 0] = 1 - p_i = q_i$, then for $X := \sum_{i=1}^n x_i$ and $\mu := \mathbb{E}[X] = \sum_{i=1}^n p_i$ the following holds:

$$\textit{for all } 0 < \delta < 1\textit{: } Pr[X \leq (1-\delta)\mu] \leq e^{-\mu\delta^2/2} \; .$$

Theorem 8.18. *The asymptotic failure probability of the Majority quorum system is 0.*

Proof. In a Majority quorum system each quorum contains exactly $\lfloor \frac{n}{2} \rfloor + 1$ nodes and each subset of nodes with cardinality $\lfloor \frac{n}{2} \rfloor + 1$ forms a quorum. The Majority quorum system fails, if only $\lfloor \frac{n}{2} \rfloor$ nodes work. Otherwise there is at least one quorum available. In order to calculate the failure probability we define the following random variables:

$$x_i = \begin{cases} 1, & \text{if node } i \text{ works, happens with probability } p \\ 0, & \text{if node } i \text{ fails, happens with probability } q = 1-p \end{cases}$$

and $X := \sum_{i=1}^n x_i$, with $\mu = np$,

whereas X corresponds to the number of working nodes. To estimate the probability that the number of working nodes is less than $\lfloor \frac{n}{2} \rfloor + 1$ we will make use of the Chernoff inequality from above. By setting $\delta = 1 - \frac{1}{2p}$ we obtain $F_P(\mathcal{S}) = Pr[X \leq \lfloor \frac{n}{2} \rfloor] \leq Pr[X \leq \frac{n}{2}] = Pr[X \leq (1-\delta)\mu]$.

With $\delta = 1 - \frac{1}{2p}$ we have $0 < \delta \leq 1/2$ due to $1/2 < p \leq 1$. Thus, we can use the Chernoff bound and get $F_P(\mathcal{S}) \leq e^{-\mu\delta^2/2} \in e^{-\Omega(n)}$. □

Theorem 8.19. *The asymptotic failure probability of the Grid quorum system is* 1.

Proof. Consider the $n = d \cdot d$ nodes to be arranged in a $d \times d$ grid. A quorum always contains one full row. In this estimation we will make use of the Bernoulli inequality which states that for all $n \in \mathbb{N}, x \geq -1 : (1+x)^n \geq 1 + nx$.

The system fails, if in each row at least one node fails (which happens with probability $1 - p^d$ for a particular row, as all nodes work with probability p^d). Therefore we can bound the failure probability from below with:

$F_p(\mathcal{S}) \geq Pr[\text{at least one failure per row}] = (1-p^d)^d \geq 1 - dp^d \underset{n\to\infty}{\longrightarrow} 1.$ □

Remarks:

- Now we have a quorum system with optimal load (the Grid) and one with fault-tolerance (Majority), but what if we want both?

Definition 8.20 (B-Grid quorum system)**.** *Consider $n = dhr$ nodes, arranged in a rectangular grid with $h \cdot r$ rows and d columns. Each group of r rows is a band, and r elements in a column restricted to a band are called a mini-column. A quorum consists of one mini-column in every band and one element from each mini-column of one band; thus every quorum has $d + hr - 1$ elements. The* ***B-Grid*** *quorum system consists of all such quorums.*

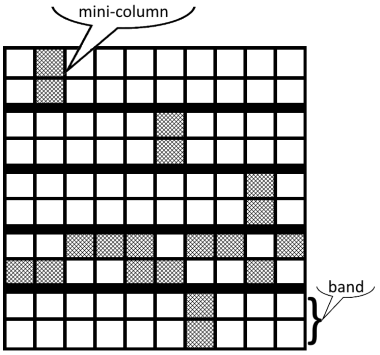


Figure 8.21: A B-Grid quorum system with $n = 100$ nodes, $d = 10$ columns, $h \cdot r = 10$ rows, $h = 5$ bands, and $r = 2$. The depicted quorum has a $d + hr - 1 = 10 + 5 \cdot 2 - 1 = 19$ nodes. If the access strategy Z is chosen uniformly, then we have a work of $d + hr - 1$ and a load of $\frac{d+hr-1}{n}$. By setting $d = \sqrt{n}$ and $r = \log n$, we obtain a work of $\Theta(\sqrt{n})$ and a load of $\Theta(1/\sqrt{n})$.

Theorem 8.22. *The asymptotic failure probability of the B-Grid quorum system is* 0.

Proof. Suppose $n = dhr$ and the elements are arranged in a grid with d columns and $h \cdot r$ rows. The B-Grid quorum system does fail if in each band a complete mini-column fails, because then it is not possible to choose a band where in each mini-column an element is still working. It also fails if in a band an element in each mini-column fails. Those events may not be independent of each other, but with the help of the union bound, we can upper bound the failure probability with the following equation:

$$\begin{aligned} F_p(\mathcal{S}) &\leq Pr[\text{in every band a complete mini-column fails}] \\ &\quad + Pr[\text{in a band at least one element of every m.-col. fails}] \\ &\leq (d(1-p)^r)^h + h(1-p^r)^d \end{aligned}$$

We use $d = \sqrt{n}, r = \ln d$, and $0 \leq (1-p) \leq 1/3$. Using $n^{\ln x} = x^{\ln n}$, we have $d(1-p)^r \leq d \cdot d^{\ln 1/3} \approx d^{-0.1}$, and hence for large enough d the whole first term is bounded from above by $d^{-0.1h} \ll 1/d^2 = 1/n$.

Regarding the second term, we have $p \geq 2/3$, and $h = d/\ln d < d$. Hence we can bound the term from above by $d(1 - d^{\ln 2/3})^d \approx d(1 - d^{-0.4})^d$. Using $(1 + t/n)^n \leq e^t$, we get (again, for large enough d) an upper bound of $d(1 - d^{-0.4})^d = d(1 - d^{0.6}/d)^d \leq d \cdot e^{-d^{0.6}} = d^{(-d^{0.6}/\ln d)+1} \ll d^{-2} = 1/n$. In total, we have $F_p(\mathcal{S}) \in O(1/n)$. □

8.4 Byzantine Quorum Systems

While failed nodes are bad, they are still easy to deal with: just access another quorum where all nodes can respond! Byzantine nodes make life more difficult however, as they can pretend to be a regular node, i.e., one needs more sophisticated methods to deal with them. We need to ensure that the intersection of two quorums always contains a non-byzantine (correct) node and furthermore, the byzantine nodes should not be allowed to infiltrate every quorum. In

	Singleton	**Majority**	**Grid**	**B-Grid***
Work	$\mathbf{1}$	$> n/2$	$\Theta\left(\sqrt{n}\right)$	$\Theta\left(\sqrt{n}\right)$
Load	1	$> 1/2$	$\boldsymbol{\Theta\left(1/\sqrt{n}\right)}$	$\boldsymbol{\Theta\left(1/\sqrt{n}\right)}$
Resilience	0	$< \mathbf{n/2}$	$\Theta\left(\sqrt{n}\right)$	$\Theta\left(\sqrt{n}\right)$
F. Prob.**	$1-p$	$\rightarrow \mathbf{0}$	$\rightarrow 1$	$\rightarrow \mathbf{0}$

Table 8.23: Overview of the different quorum systems regarding resilience, work, load, and their asymptotic failure probability. The best entries in each row are set in bold.

* Setting $d = \sqrt{n}$ and $r = \log n$

**Assuming prob. $q = (1-p)$ is constant but significantly less than 1/2

this section we study three counter-measures of increasing strength, and their implications on the load of quorum systems.

Definition 8.24 (f-disseminating)**.** *A quorum system $\mathcal{S}$ is f-**disseminating** if* (1) *the intersection of two different quorums always contains* $f+1$ *nodes, and* (2) *for any set of f byzantine nodes, there is at least one quorum without byzantine nodes.*

Remarks:

- Thanks to (2), even with f byzantine nodes, the byzantine nodes cannot stop all quorums by just pretending to have crashed. At least one quorum will survive. We will also keep this assumption for the upcoming more advanced byzantine quorum systems.
- Byzantine nodes can also do something worse than crashing - they could falsify data! Nonetheless, due to (1), there is at least one non-byzantine node in every quorum intersection. If the data is self-verifying by, e.g., authentication, then this one node is enough.
- If the data is not self-verifying, then we need another mechanism.

Definition 8.25 (f-masking)**.** *A quorum system $\mathcal{S}$ is f-**masking** if* (1) *the intersection of two different quorums always contains* $2f+1$ *nodes, and* (2) *for any set of f byzantine nodes, there is at least one quorum without byzantine nodes.*

Remarks:

- Note that except for the second condition, an f-masking quorum system is the same as a $2f$-disseminating system. The idea is that the non-byzantine nodes (at least $f+1$) can outvote the byzantine ones (at most f), but only if all non-byzantine nodes are up-to-date!
- This raises an issue not covered yet in this chapter. If we access some quorum and update its values, this change still has to be disseminated to the other nodes in the byzantine quorum system. Opaque quorum systems deal with this issue, which are discussed at the end of this section.

- f-disseminating quorum systems need more than $3f$ nodes and f-masking quorum systems need more than $4f$ nodes. Essentially, the quorums may not contain too many nodes, and the different intersection properties lead to the different bounds.

Theorem 8.26. *Let $\mathcal{S}$ be a f-disseminating quorum system. Then $L(\mathcal{S}) \geq \sqrt{(f+1)/n}$ holds.*

Theorem 8.27. *Let $\mathcal{S}$ be a f-masking quorum system. Then $L(\mathcal{S}) \geq \sqrt{(2f+1)/n}$ holds.*

Proofs of Theorems 8.26 and 8.27. The proofs follow the proof of Theorem 8.6, by observing that now not just one element is accessed from a minimal quorum, but $f+1$ or $2f+1$, respectively. □

Definition 8.28 (f-masking Grid quorum system)**.** *A **f-masking Grid** quorum system is constructed as the grid quorum system, but each quorum contains one full column and $f+1$ rows of nodes, with $2f+1 \leq \sqrt{n}$.*

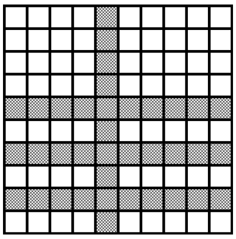


Figure 8.29: An example how to choose a quorum in the f-masking Grid with $f = 2$, i.e., $2 + 1 = 3$ rows. The load is in $\Theta(f/\sqrt{n})$ when the access strategy is chosen to be uniform. Two quorums overlap by their columns intersecting each other's rows, i.e., they overlap in at least $2f + 2$ nodes.

Remarks:

- The f-masking Grid nearly hits the lower bound for the load of f-masking quorum systems, but not quite. A small change and we will be optimal asymptotically.

Definition 8.30 (M-Grid quorum system)**.** *The M-**Grid** quorum system is constructed as the grid quorum as well, but each quorum contains $\sqrt{f+1}$ rows and $\sqrt{f+1}$ columns of nodes, with $f \leq \frac{\sqrt{n-1}}{2}$.*

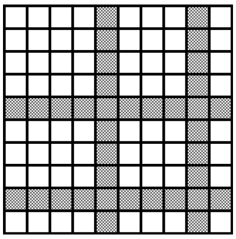


Figure 8.31: An example how to choose a quorum in the M-Grid with $f = 3$, i.e., 2 rows and 2 columns. The load is in $\Theta(\sqrt{f/n})$ when the access strategy is chosen to be uniform. Two quorums overlap with each row intersecting each other's column, i.e., $2\sqrt{f+1}^2 = 2f + 2$ nodes.

Corollary 8.32. *The f-masking Grid quorum system and the M-Grid quorum system are f-masking quorum systems.*

Remarks:

- We achieved nearly the same load as without byzantine nodes! However, as mentioned earlier, what happens if we access a quorum that is not up-to-date, except for the intersection with an up-to-date quorum? Surely we can fix that as well without too much loss?
- This property will be handled in the last part of this chapter by *opaque* quorum systems. It will ensure that the number of correct up-to-date nodes accessed will be larger than the number of out-of-date nodes combined with the byzantine nodes in the quorum (cf. (8.33.1)).

Definition 8.33 (f-opaque quorum system)**.** *A quorum system $\mathcal{S}$ is f-**opaque** if the following two properties hold for any set of f byzantine nodes F and any two different quorums Q_1, Q_2:*

$$|(Q_1 \cap Q_2) \setminus F| > |(Q_2 \cap F) \cup (Q_2 \setminus Q_1)| \tag{8.33.1}$$

$$(F \cap Q) = \emptyset \text{ for some } Q \in \mathcal{S} \tag{8.33.2}$$

Theorem 8.35. *Let $\mathcal{S}$ be a f-opaque quorum system. Then, $n > 5f$.*

Proof. Due to (8.33.2), there exists a quorum Q_1 with size at most $n - f$. With (8.33.1), $|Q_1| > f$ holds. Let F_1 be a set of f (byzantine) nodes $F_1 \subset Q_1$, and with (8.33.2), there exists a $Q_2 \subset V \setminus F_1$. Thus, $|Q_1 \cap Q_2| \leq n - 2f$. With (8.33.1), $|Q_1 \cap Q_2| > f$ holds. Thus, one could choose f (byzantine) nodes F_2 with $F_2 \subset (Q_1 \cap Q_2)$. Using (8.33.1) one can bound $n - 3f$ from below: $n - 3f > |(Q_2 \cap Q_1)| - |F_2| \geq |(Q_2 \cap Q_1) \cup (Q_1 \cap F_2)| \geq |F_1| + |F_2| = 2f.$ □

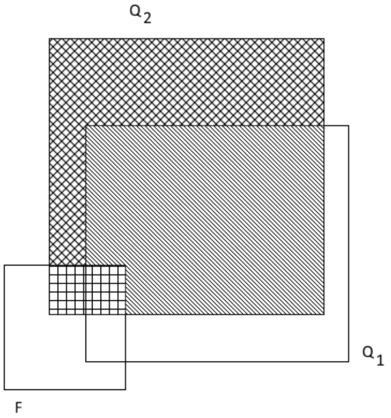


Figure 8.34: Intersection properties of an opaque quorum system. Equation (8.33.1) ensures that the set of non-byzantine nodes in the intersection of Q_1, Q_2 is larger than the set of out of date nodes, even if the byzantine nodes "team up" with those nodes. Thus, the correct up to date value can always be recognized by a majority voting.

Remarks:

- One can extend the Majority quorum system to be f-opaque by setting the size of each quorum to contain $\lceil(2n + 2f)/3\rceil$ nodes. Then its load is $1/n\,\lceil(2n + 2f)/3\rceil \approx 2/3 + 2f/3n \geq 2/3$.
- Can we do much better? Sadly, no...

Theorem 8.36. *Let $\mathcal{S}$ be a f-opaque quorum system. Then $L(\mathcal{S}) \geq 1/2$ holds.*

Proof. Equation (8.33.1) implies that for $Q_1, Q_2 \in \mathcal{S}$, the intersection of both Q_1, Q_2 is at least half their size, i.e., $|(Q_1 \cap Q_2)| \geq |Q_1|/2$. Let $\mathcal{S}$ consist of quorums $Q_1, Q_2, \ldots$. The load induced by an access strategy Z on Q_1 is:

$$\sum_{v \in Q_1} \sum_{v \in Q_i} L_Z(Q_i) = \sum_{Q_i} \sum_{v \in (Q_1 \cap Q_i)} L_Z(Q_i) \geq \sum_{Q_i} (|Q_1|/2)\ L_Z(Q_i) = |Q_1|/2\ .$$

Using the pigeonhole principle, there must be at least one node in Q_1 with load of at least $1/2$. □

Chapter Notes

Historically, a quorum is the minimum number of members of a deliberative body necessary to conduct the business of that group. Their use has inspired the introduction of quorum systems in computer science since the late 1970s/early 1980s. Early work focused on Majority quorum systems [Lam78, Gif79, Tho79], with the notion of minimality introduced shortly after [GB85]. The Grid quorum system was first considered in [Mae85], with the B-Grid being introduced in [NW94]. The latter article and [PW95] also initiated the study of load and resilience.

The f-masking Grid quorum system and opaque quorum systems are from [MR98], and the M-Grid quorum system was introduced in [MRW97]. Both papers also mark the start of the formal study of Byzantine quorum systems. The f-masking and the M-Grid have asymptotic failure probabilities of 1, more complex systems with better values can be found in these papers as well.

Quorum systems have also been extended to cope with nodes dynamically leaving and joining, see, e.g., the dynamic paths quorum system in [NW05].

For a further overview on quorum systems, we refer to the book by Vukolić [Vuk12] and the article by Merideth and Reiter [MR10].

This chapter was written in collaboration with Klaus-Tycho Förster.

Bibliography

[GB85] Hector Garcia-Molina and Daniel Barbará. How to assign votes in a distributed system. *J. ACM*, 32(4):841–860, 1985.

[Gif79] David K. Gifford. Weighted voting for replicated data. In Michael D. Schroeder and Anita K. Jones, editors, *Proceedings of the Seventh Symposium on Operating System Principles, SOSP 1979, Asilomar Conference Grounds, Pacific Grove, California, USA, 10-12, December 1979*, pages 150–162. ACM, 1979.

[Lam78] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978.

[Mae85] Mamoru Maekawa. A square root N algorithm for mutual exclusion in decentralized systems. *ACM Trans. Comput. Syst.*, 3(2):145–159, 1985.

[MR98] Dahlia Malkhi and Michael K. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):203–213, 1998.

[MR10] Michael G. Merideth and Michael K. Reiter. Selected results from the latest decade of quorum systems research. In Bernadette Charron-Bost, Fernando Pedone, and André Schiper, editors, *Replication: Theory and Practice*, volume 5959 of *Lecture Notes in Computer Science*, pages 185–206. Springer, 2010.

[MRW97] Dahlia Malkhi, Michael K. Reiter, and Avishai Wool. The load and availability of byzantine quorum systems. In James E. Burns and Hagit Attiya, editors, *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing, Santa Barbara, California, USA, August 21-24, 1997*, pages 249–257. ACM, 1997.

[NW94] Moni Naor and Avishai Wool. The load, capacity and availability of quorum systems. In *35th Annual Symposium on Foundations of Computer Science, Santa Fe, New Mexico, USA, 20-22 November 1994*, pages 214–225. IEEE Computer Society, 1994.

[NW05] Moni Naor and Udi Wieder. Scalable and dynamic quorum systems. *Distributed Computing*, 17(4):311–322, 2005.

[PW95] David Peleg and Avishai Wool. The availability of quorum systems. *Inf. Comput.*, 123(2):210–223, 1995.

[Tho79] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Trans. Database Syst.*, 4(2):180–209, 1979.

[Vuk12] Marko Vukolic. *Quorum Systems: With Applications to Storage and Consensus*. Synthesis Lectures on Distributed Computing Theory. Morgan & Claypool Publishers, 2012.

Chapter 9

Eventual Consistency & Bitcoin

How would you implement an ATM? Does the following implementation work satisfactorily?

Algorithm 9.1 Naïve ATM

```
1: ATM makes withdrawal request to bank
2: ATM waits for response from bank
3: if balance of customer sufficient then
4:     ATM dispenses cash
5: else
6:     ATM displays error
7: end if
```

Remarks:

- A connection problem between the bank and the ATM may block Algorithm 9.1 in Line 2.
- A *network partition* is a failure where a network splits into at least two parts that cannot communicate with each other. Intuitively any non-trivial distributed system cannot proceed during a partition *and* maintain consistency. In the following we introduce the tradeoff between consistency, availability and partition tolerance.
- There are numerous causes for partitions to occur, e.g., physical disconnections, software errors, or incompatible protocol versions. From the point of view of a node in the system, a partition is similar to a period of sustained message loss.

9.1 Consistency, Availability and Partitions

Definition 9.2 (Consistency). *All nodes in the system agree on the current state of the system.*

Definition 9.3 (Availability). *The system is operational and instantly processing incoming requests.*

Definition 9.4 (Partition Tolerance). *Partition tolerance is the ability of a distributed system to continue operating correctly even in the presence of a network partition.*

Theorem 9.5 (CAP Theorem). *It is impossible for a distributed system to simultaneously provide Consistency, Availability and Partition Tolerance. A distributed system can satisfy any two of these but not all three.*

Proof. Assume two nodes, sharing some state. The nodes are in different partitions, i.e., they cannot communicate. Assume a request wants to update the state and contacts a node. The node may either: 1) update its local state, resulting in inconsistent states, or 2) not update its local state, i.e., the system is no longer available for updates. □

Algorithm 9.6 Partition tolerant and available ATM

```
if bank reachable then
  Synchronize local view of balances between ATM and bank
  if balance of customer insufficient then
    ATM displays error and aborts user interaction
  end if
end if
ATM dispenses cash
ATM logs withdrawal for synchronization
```

Remarks:

- Algorithm 9.6 is partition tolerant and available since it continues to process requests even when the bank is not reachable.
- The ATM's local view of the balances may diverge from the balances as seen by the bank, therefore consistency is no longer guaranteed.
- The algorithm will synchronize any changes it made to the local balances back to the bank once connectivity is re-established. This is known as eventual consistency.

Definition 9.7 (Eventual Consistency). *If no new updates to the shared state are issued, then eventually the system is in a quiescent state, i.e., no more messages need to be exchanged between nodes, and the shared state is consistent.*

Remarks:

- Eventual consistency is a form of *weak consistency.*
- Eventual consistency guarantees that the state is eventually agreed upon, but the nodes may disagree temporarily.
- During a partition, different updates may semantically conflict with each other. A *conflict resolution* mechanism is required to resolve the conflicts and allow the nodes to eventually agree on a common state.

- One example of eventual consistency is the Bitcoin cryptocurrency system.

9.2 Bitcoin

Definition 9.8 (Bitcoin Network)**.** *The Bitcoin network is a randomly connected overlay network of a few thousand* ***nodes****, controlled by a variety of owners. All nodes perform the same operations, i.e., it is a homogenous network and without central control.*

Remarks:

- The lack of structure is intentional: it ensures that an attacker cannot strategically position itself in the network and manipulate the information exchange. Information is exchanged via a simple broadcasting protocol.

Definition 9.9 (Address)**.** *Users may generate any number of private keys, from which a public key is then derived. An address is derived from a public key and may be used to identify the recipient of funds in Bitcoin. The private/public key pair is used to uniquely identify the owner of funds of an address.*

Remarks:

- The terms public key and address are often used interchangeably, since both are public information. The advantage of using an address is that its representation is shorter than the public key.

- It is hard to link addresses to the user that controls them, hence Bitcoin is often referred to as being *pseudonymous*.

- Not every user needs to run a fully validating node, and end-users will likely use a lightweight client that only temporarily connects to the network.

- The Bitcoin network collaboratively tracks the balance in bitcoins of each address.

- The address is composed of a network identifier byte, the hash of the public key and a checksum. It is commonly stored in base 58 encoding, a custom encoding similar to base 64 with some ambiguous symbols removed, e.g., lowercase letter "l" since it is similar to the number "1".

- The hashing algorithm produces addresses of size 20 bytes. This means that there are 2^{160} distinct addresses. It might be tempting to brute force a target address, however at one billion trials per second one still requires approximately 2^{45} years in expectation to find a matching private/public key pair. Due to the birthday paradox the odds improve if instead of brute forcing a single address we attempt to brute force any address. While the odds of a successful trial increase with the number of addresses, lookups become more costly.

Definition 9.10 (Output)**.** *An output is a tuple consisting of an amount of bitcoins and a spending condition. Most commonly the spending condition requires a valid signature associated with the private key of an address.*

Remarks:

- Spending conditions are scripts that offer a variety of options. Apart from a single signature, they may include conditions that require the result of a simple computation, or the solution to a cryptographic puzzle.
- Outputs exist in two states: unspent and spent. Any output can be spent at most once. The address balance is the sum of bitcoin amounts in unspent outputs that are associated with the address.
- The set of unspent transaction outputs (UTXOs) and some additional global parameters are the shared state of Bitcoin. Every node in the Bitcoin network holds a complete replica of that state. Local replicas may temporarily diverge, but consistency is eventually re-established.

Definition 9.11 (Input)**.** *An input is a tuple consisting of a reference to a previously created output and arguments (signature) to the spending condition, proving that the transaction creator has the permission to spend the referenced output.*

Definition 9.12 (Transaction)**.** *A transaction is a data structure that describes the transfer of bitcoins from spenders to recipients. The transaction consists of a number of inputs and new outputs. The inputs result in the referenced outputs spent (removed from the UTXO), and the new outputs being added to the UTXO.*

Remarks:

- Inputs reference the output that is being spent by a (h, i)-tuple, where h is the hash of the transaction that created the output, and i specifies the index of the output in that transaction.
- Transactions are broadcast in the Bitcoin network and processed by every node that receives them.

Algorithm 9.13 Node Receives Transaction

```
Receive transaction $t$
for each input $(h, i)$ in $t$ do
  if output $(h, i)$ is not in local UTXO or signature invalid then
    Drop $t$ and stop
  end if
end for
if sum of values of inputs $<$ sum of values of new outputs then
  Drop $t$ and stop
end if
for each input $(h, i)$ in $t$ do
  Remove $(h, i)$ from local UTXO
end for
Append $t$ to local history
Forward $t$ to neighbors in the Bitcoin network
```

Remarks:

- Note that the effect of a transaction on the state is deterministic. In other words if all nodes receive the same set of transactions in the same order (Definition 2.8), then the state across nodes is consistent.
- The outputs of a transaction may assign less than the sum of inputs, in which case the difference is called the transaction *fee*. The fee is used to incentivize other participants in the system (see Definition 9.19)
- Notice that so far we only described a local acceptance policy. Nothing prevents nodes to locally accept different transactions that spend the same output.
- Transactions are in one of two states: unconfirmed or confirmed. Incoming transactions from the broadcast are unconfirmed and added to a pool of transactions called the *memory pool*.

Definition 9.14 (Doublespend)**.** *A doublespend is a situation in which multiple transactions attempt to spend the same output. Only one transaction can be valid since outputs can only be spent once. When nodes accept different transactions in a doublespend, the shared state becomes inconsistent.*

Remarks:

- Doublespends may occur naturally, e.g., if outputs are co-owned by multiple users. However, often doublespends are intentional – we call these doublespend-attacks: In a transaction, an attacker pretends to transfer an output to a victim, only to doublespend the same output in another transaction back to itself.
- Doublespends can result in an inconsistent state since the validity of transactions depends on the order in which they arrive. If two conflicting transactions are seen by a node, the node considers the first to be valid, see Algorithm 9.13. The second transaction is invalid

since it tries to spend an output that is already spent. The order in which transactions are seen, may not be the same for all nodes, hence the inconsistent state.

- If doublespends are not resolved, the shared state diverges. Therefore a conflict resolution mechanism is needed to decide which of the conflicting transactions is to be confirmed (accepted by everybody), to achieve eventual consistency.

Definition 9.15 (Proof-of-Work). *Proof-of-Work (PoW) is a mechanism that allows a party to prove to another party that a certain amount of computational resources has been utilized for a period of time. A function $\mathcal{F}_d(c, x) \to \{true, false\}$, where difficulty d is a positive number, while challenge c and nonce x are usually bit-strings, is called a Proof-of-Work function if it has following properties:*

1. *$\mathcal{F}_d(c, x)$ is fast to compute if d, c, and x are given.*
2. *For fixed parameters d and c, finding x such that $\mathcal{F}_d(c, x) = true$ is computationally difficult but feasible. The difficulty d is used to adjust the time to find such an x.*

Definition 9.16 (Bitcoin PoW function). *The Bitcoin PoW function is given by*

$$\mathcal{F}_d(c, x) \to \text{SHA256}(\text{SHA256}(c|x)) < \frac{2^{224}}{d}.$$

Remarks:

- This function concatenates the challenge c and nonce x, and hashes them twice using SHA256. The output of SHA256 is a cryptographic hash with a numeric value in $\{0, \ldots, 2^{256} - 1\}$ which is compared to a target value $\frac{2^{224}}{d}$, which gets smaller with increasing difficulty.
- SHA256 is a cryptographic hash function with pseudorandom output. No better algorithm is known to find a nonce x such that the function $\mathcal{F}_d(c, x)$ returns true than simply iterating over possible inputs. This is by design to make it difficult to find such an input, but simple to verify the validity once it has been found.
- If the PoW functions of all nodes had the same challenge, the fastest node would always win. However, as we will see in Definition 9.19, each node attempts to find a valid nonce for a node-specific challenge.

Definition 9.17 (Block). *A block is a data structure used to communicate incremental changes to the local state of a node. A block consists of a list of transactions, a reference to a previous block and a nonce. A block lists some transactions the block creator ("miner") has accepted to its memory pool since the previous block. A node finds and broadcasts a block when it finds a valid nonce for its PoW function.*

Algorithm 9.18 Node Finds Block

```
Nonce $x = 0$, challenge $c$, difficulty $d$, previous block $b_{t-1}$
repeat
    $x = x + 1$
until $\mathcal{F}_d(c, x) = true$
Broadcast block $b_t = (memory\ pool, b_{t-1}, x)$
```

Remarks:

- With their reference to a previous block, the blocks build a tree, rooted in the so called *genesis block*.
- The primary goal for using the PoW mechanism is to adjust the rate at which blocks are found in the network, giving the network time to synchronize on the latest block. Bitcoin sets the difficulty so that globally a block is created about every 10 minutes in expectation.
- Finding a block allows the finder to impose the transactions in its local memory pool to all other nodes. Upon receiving a block, all nodes roll back any local changes since the previous block and apply the new block's transactions.
- Transactions contained in a block are said to be *confirmed* by that block.

Definition 9.19 (Reward Transaction)**.** *The first transaction in a block is called the reward transaction. The block's miner is rewarded for confirming transactions by allowing it to mint new coins. The reward transaction has a dummy input, and the sum of outputs is determined by a fixed subsidy plus the sum of the fees of transactions confirmed in the block.*

Remarks:

- A reward transaction is the sole exception to the rule that the sum of inputs must be at least the sum of outputs.
- The number of bitcoins that are minted by the reward transaction and assigned to the miner is determined by a subsidy schedule that is part of the protocol. Initially the subsidy was 50 bitcoins for every block, and it is being halved every 210,000 blocks, or 4 years in expectation. Due to the halving of the block reward, the total amount of bitcoins in circulation never exceeds 21 million bitcoins.
- It is expected that the cost of performing the PoW to find a block, in terms of energy and infrastructure, is close to the value of the reward the miner receives from the reward transaction in the block.

Definition 9.20 (Blockchain)**.** *The longest path from the genesis block, i.e., root of the tree, to a leaf is called the blockchain. The blockchain acts as a consistent transaction history on which all nodes eventually agree.*

Remarks:

- The path length from the genesis block to block b is the height h_b.
- Only the longest path from the genesis block to a leaf is a valid transaction history, since branches may contradict each other because of doublespends.
- Since only transactions in the longest path are agreed upon, miners have an incentive to append their blocks to the longest chain, thus agreeing on the current state.
- The mining incentives quickly increased the difficulty of the PoW mechanism: initially miners used CPUs to mine blocks, but CPUs were quickly replaced by GPUs, FPGAs and even application specific integrated circuits (AS-ICs) as bitcoins appreciated. This results in an equilibrium today in which only the most cost efficient miners, in terms of hardware supply and electricity, make a profit in expectation.
- If multiple blocks are mined more or less concurrently, the system is said to have *forked*. Forks happen naturally because mining is a distributed random process and two new blocks may be found at roughly the same time.

Algorithm 9.21 Node Receives Block

```
Receive block $b$
For this node the current head is block $b_{max}$ at height $h_{max}$
Connect block $b$ in the tree as child of its parent $p$ at height $h_b = h_p + 1$
if $h_b > h_{max}$ then
    $h_{max} = h_b$
    $b_{max} = b$
    Compute UTXO for the path leading to $b_{max}$
    Cleanup memory pool
end if
```

Remarks:

- Algorithm 9.21 describes how a node updates its local state upon receiving a block. Notice that, like Algorithm 9.13, this describes the local policy and may also result in node states diverging, i.e., by accepting different blocks at the same height as current head.
- Unlike extending the current path, switching paths may result in confirmed transactions no longer being confirmed, because the blocks in the new path do not include them. Switching paths is referred to as a *reorg*.
- Cleaning up the memory pool involves 1) removing transactions that were confirmed in a block in the current path, 2) removing transactions that conflict with confirmed transactions, and 3) adding transactions that were confirmed in the previous path, but are no longer confirmed in the current path.

- In order to avoid having to recompute the entire UTXO at every new block being added to the blockchain, all current implementations use data structures that store undo information about the operations applied by a block. This allows efficient switching of paths and updates of the head by moving along the path.

Theorem 9.22. *Forks are eventually resolved and all nodes eventually agree on which is the longest blockchain. The system therefore guarantees eventual consistency.*

Proof. In order for the fork to continue to exist, pairs of blocks need to be found in close succession, extending distinct branches, otherwise the nodes on the shorter branch would switch to the longer one. The probability of branches being extended almost simultaneously decreases exponentially with the length of the fork, hence there will eventually be a time when only one branch is being extended, becoming the longest branch. □

9.3 Smart Contracts

Definition 9.23 (Smart Contract)**.** *A smart contract is an agreement between two or more parties, encoded in such a way that the correct execution is guaranteed by the blockchain.*

Remarks:

- Contracts allow business logic to be encoded in Bitcoin transactions which mutually guarantee that an agreed upon action is performed. The blockchain acts as conflict mediator, should a party fail to honor an agreement.

- The use of scripts as spending conditions for outputs enables smart contracts. Scripts, together with some additional features such as timelocks, allow encoding complex conditions, specifying who may spend the funds associated with an output and when.

Definition 9.24 (Timelock)**.** *Bitcoin provides a mechanism to make transactions invalid until some time in the future:* ***timelocks****. A transaction may specify a locktime: the earliest time, expressed in either a Unix timestamp or a blockchain height, at which it may be included in a block and therefore be confirmed.*

Remarks:

- Transactions with a timelock are not released into the network until the timelock expires. It is the responsibility of the node receiving the transaction to store it locally until the timelock expires and then release it into the network.

- Transactions with future timelocks are invalid. Blocks may not include transactions with timelocks that have not yet expired, i.e., they are mined before their expiry timestamp or in a lower block than specified. If a block includes an unexpired transaction it is invalid. Upon receiving invalid transactions or blocks, nodes discard them immediately and do not forward them to their peers.

- Timelocks can be used to replace or supersede transactions: a timelocked transaction t_1 can be replaced by another transaction t_0, spending some of the same outputs, if the replacing transaction t_0 has an earlier timelock and can be broadcast in the network before the replaced transaction t_1 becomes valid.

Definition 9.25 (Singlesig and Multisig Outputs)**.** *When an output can be claimed by providing a single signature it is called a **singlesig output**. In contrast the script of **multisig outputs** specifies a set of m public keys and requires k-of-m (with $k \leq m$) valid signatures from distinct matching public keys from that set in order to be valid.*

Remarks:

- Most smart contracts begin with the creation of a 2-of-2 multisig output, requiring a signature from both parties. Once the transaction creating the multisig output is confirmed in the blockchain, both parties are guaranteed that the funds of that output cannot be spent unilaterally.

Algorithm 9.26 Parties A and B create a 2-of-2 multisig output o

```
1: $B$ sends a list $I_B$ of inputs with $c_B$ coins to $A$
2: $A$ selects its own inputs $I_A$ with $c_A$ coins
3: $A$ creates transaction $t_s\{[I_A, I_B], [o = c_A + c_B \rightarrow (A, B)]\}$
4: $A$ creates timelocked transaction $t_r\{[o], [c_A \rightarrow A, c_B \rightarrow B]\}$ and signs it
5: $A$ sends $t_s$ and $t_r$ to $B$
6: $B$ signs both $t_s$ and $t_r$ and sends them to $A$
7: $A$ signs $t_s$ and broadcasts it to the Bitcoin network
```

Remarks:

- t_s is called a *setup transaction* and is used to lock in funds into a shared account. If t_s is signed and broadcast immediately, one of the parties could not collaborate to spend the multisig output, and the funds become unspendable. To avoid a situation where the funds cannot be spent, the protocol also creates a timelocked *refund transaction* t_r which guarantees that, should the funds not be spent before the timelock expires, the funds are returned to the respective party. At no point in time one of the parties holds a fully signed setup transaction without the other party holding a fully signed refund transaction, guaranteeing that funds are eventually returned.

- Both transactions require the signature of both parties. In the case of the setup transaction because it has two inputs from A and B respectively which require individual signatures. In the case of the refund transaction the single input spending the multisig output requires both signatures being a 2-of-2 multisig output.

Algorithm 9.27 Simple Micropayment Channel from S to R with capacity c

```
$c_S = c$, $c_R = 0$
$S$ and $R$ use Algorithm 9.26 to set up output $o$ with value $c$ from $S$
Create settlement transaction $t_f\{[o], [c_S \to S, c_R \to R]\}$
while channel open and $c_R < c$ do
    In exchange for good with value $\delta$
    $c_R = c_R + \delta$
    $c_S = c_S - \delta$
    Update $t_f$ with outputs $[c_R \to R, c_S \to S]$
    $S$ signs and sends $t_f$ to $R$
end while
$R$ signs last $t_f$ and broadcasts it
```

Remarks:

- Algorithm 9.27 implements a Simple Micropayment Channel, a smart contract that is used for rapidly adjusting micropayments from a spender to a recipient. Only two transactions are ever broadcast and inserted into the blockchain: the setup transaction t_s and the last settlement transaction t_f. There may have been any number of updates to the settlement transaction, transferring ever more of the shared output to the recipient.

- The number of bitcoins c used to fund the channel is also the maximum total that may be transferred over the simple micropayment channel.

- At any time the recipient R is guaranteed to eventually receive the bitcoins, since she holds a fully signed settlement transaction, while the spender only has partially signed ones.

- The simple micropayment channel is intrinsically unidirectional. Since the recipient may choose any of the settlement transactions in the protocol, she will use the one with maximum payout for her. If we were to transfer bitcoins back, we would be reducing the amount paid out to the recipient, hence she would choose not to broadcast that transaction.

9.4 Weak Consistency

Eventual consistency is only one form of weak consistency. A number of different tradeoffs between partition tolerance and consistency exist in literature.

Definition 9.28 (Monotonic Read Consistency)**.** *If a node u has seen a particular value of an object, any subsequent accesses of u will never return any older values.*

Remarks:

- Users are annoyed if they receive a notification about a comment on an online social network, but are unable to reply because the web interface does not show the same notification yet. In this case the notification acts as the first read operation, while looking up the comment on the web interface is the second read operation.

Definition 9.29 (Monotonic Write Consistency)**.** *A write operation by a node on a data item is completed before any successive write operation by the same node (i.e., system guarantees to serialize writes by the same node).*

Remarks:

- The ATM must replay all operations in order, otherwise it might happen that an earlier operation overwrites the result of a later operation, resulting in an inconsistent final state.

Definition 9.30 (Read-Your-Write Consistency)**.** *After a node u has updated a data item, any later reads from node u will never see an older value.*

Definition 9.31 (Causal Relation)**.** *The following pairs of operations are said to be causally related:*

- *Two writes by the same node to different variables.*
- *A read followed by a write of the same node.*
- *A read that returns the value of a write from any node.*
- *Two operations that are transitively related according to the above conditions.*

Remarks:

- The first rule ensures that writes by a single node are seen in the same order. For example if a node writes a value in one variable and then signals that it has written the value by writing in another variable. Another node could then read the signalling variable but still read the old value from the first variable, if the two writes were not causally related.

Definition 9.32 (Causal Consistency)**.** *A system provides causal consistency if operations that potentially are causally related are seen by every node of the system in the same order. Concurrent writes are not causally related, and may be seen in different orders by different nodes.*

Chapter Notes

The CAP theorem was first introduced by Fox and Brewer [FB99], although it is commonly attributed to a talk by Eric Brewer [Bre00]. It was later proven by Gilbert and Lynch [GL02] for the asynchronous model. Gilbert and Lynch also showed how to relax the consistency requirement in a partially synchronous system to achieve availability and partition tolerance.

Bitcoin was introduced in 2008 by Satoshi Nakamoto [Nak08]. Nakamoto is thought to be a pseudonym used by either a single person or a group of people; it is still unknown who invented Bitcoin, giving rise to speculation and conspiracy theories. Among the plausible theories are noted cryptographers Nick Szabo [Big13] and Hal Finney [Gre14]. The first Bitcoin client was published shortly after the paper and the first block was mined on January 3, 2009. The genesis block contained the headline of the release date's The Times issue "*The Times 03/Jan/2009 Chancellor on brink of second bailout for banks*", which serves as proof that the genesis block has been indeed mined on that date, and that no one had mined before that date. The quote in the genesis block is also thought to be an ideological hint: Bitcoin was created in a climate of financial crisis, induced by rampant manipulation by the banking sector, and Bitcoin quickly grew in popularity in anarchic and libertarian circles. The original client is nowadays maintained by a group of independent core developers and remains the most used client in the Bitcoin network.

Central to Bitcoin is the resolution of conflicts due to doublespends, which is solved by waiting for transactions to be included in the blockchain. This however introduces large delays for the confirmation of payments which are undesirable in some scenarios in which an immediate confirmation is required. Karame et al. [KAC12] show that accepting unconfirmed transactions leads to a non-negligible probability of being defrauded as a result of a doublespending attack. This is facilitated by *information eclipsing* [DW13], i.e., that nodes do not forward conflicting transactions, hence the victim does not see both transactions of the doublespend. Bamert et al. [BDE+13] showed that the odds of detecting a doublespending attack in real-time can be improved by connecting to a large sample of nodes and tracing the propagation of transactions in the network.

Bitcoin does not scale very well due to its reliance on confirmations in the blockchain. A copy of the entire transaction history is stored on every node in order to bootstrap joining nodes, which have to reconstruct the transaction history from the genesis block. Simple micropayment channels were introduced by Hearn and Spilman [HS12] and may be used to bundle multiple transfers between two parties but they are limited to transferring the funds locked into the channel once. Recently Duplex Micropayment Channels [DW15] and the Lightning Network [PD15] have been proposed to build bidirectional micropayment channels in which the funds can be transferred back and forth an arbitrary number of times, greatly increasing the flexibility of Bitcoin transfers and enabling a number of features, such as micropayments and routing payments between any two endpoints.

This chapter was written in collaboration with Christian Decker.

Bibliography

[BDE+13] Tobias Bamert, Christian Decker, Lennart Elsen, Samuel Welten, and Roger Wattenhofer. Have a snack, pay with bitcoin. In *IEEE Internation Conference on Peer-to-Peer Computing (P2P), Trento, Italy*, 2013.

[Big13] John Biggs. Who is the real satoshi nakamoto? one researcher may have found the answer. http://on.tcrn.ch/l/R0vA, 2013.

[Bre00] Eric A. Brewer. Towards robust distributed systems. In *Symposium on Principles of Distributed Computing (PODC)*. ACM, 2000.

[DW13] Christian Decker and Roger Wattenhofer. Information propagation in the bitcoin network. In *IEEE International Conference on Peer-to-Peer Computing (P2P), Trento, Italy*, September 2013.

[DW15] Christian Decker and Roger Wattenhofer. A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels. In *Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2015.

[FB99] Armando Fox and Eric Brewer. Harvest, yield, and scalable tolerant systems. In *Hot Topics in Operating Systems*. IEEE, 1999.

[GL02] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 2002.

[Gre14] Andy Greenberg. Nakamoto's neighbor: My hunt for bitcoin's creator led to a paralyzed crypto genius. http://onforb.es/1rvyecq, 2014.

[HS12] Mike Hearn and Jeremy Spilman. Contract: Rapidly adjusting micro-payments. https://en.bitcoin.it/wiki/Contract, 2012. Last accessed on November 11, 2015.

[KAC12] G.O. Karame, E. Androulaki, and S. Capkun. Two Bitcoins at the Price of One? Double-Spending Attacks on Fast Payments in Bitcoin. In *Conference on Computer and Communication Security (CCS)*, 2012.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. https://bitcoin.org/bitcoin.pdf, 2008.

[PD15] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network. 2015.

Chapter 10

Game Theory

"Game theory is a sort of umbrella or 'unified field' theory for the rational side of social science, where 'social' is interpreted broadly, to include human as well as non-human players (computers, animals, plants)."

– Robert Aumann, 1987

10.1 Introduction

In this chapter we look at a distributed system from a different perspective. Nodes no longer have a common goal, but are *selfish*. The nodes are not byzantine (actively malicious), instead they try to benefit from a distributed system – possibly without contributing.

Game theory attempts to mathematically capture behavior in strategic situations, in which an individual's success depends on the choices of others.

Remarks:

- Examples of potentially selfish behavior are file sharing or TCP. If a packet is dropped, then most TCP implementations interpret this as a congested network and alleviate the problem by reducing the speed at which packets are sent. What if a selfish TCP implementation will not reduce its speed, but instead transmit each packet twice?

- We start with one of the most famous games to introduce some definitions and concepts of game theory.

10.2 Prisoner's Dilemma

A team of two prisoners (players u and v) are being questioned by the police. They are both held in solitary confinement and cannot talk to each other. The prosecutors offer a bargain to each prisoner: snitch on the other prisoner to reduce your prison sentence.

u v		Player u	
		Cooperate	Defect
Player v	Cooperate	1 1	0 3
	Defect	3 0	2 2

Table 10.1: The prisoner's dilemma game as a matrix.

- If both of them stay silent (*cooperate*), both will be sentenced to one year of prison on a lesser charge.
- If both of them testify against their fellow prisoner (*defect*), the police has a stronger case and they will be sentenced to two years each.
- If player u defects and the player v cooperates, then player u will go free (snitching pays off) and player v will have to go to jail for three years; and vice versa.
- This two player game can be represented as a matrix, see Table 10.1.

Definition 10.2 (game)**.** *A game requires at least two rational players, and each player can choose from at least two options (**strategies**). In every possible outcome (**strategy profile**) each player gets a certain payoff (or cost). The payoff of a player depends on the strategies of the other players.*

Definition 10.3 (social optimum)**.** *A strategy profile is called social optimum (SO) if and only if it minimizes the sum of all costs (or maximizes payoff).*

Remarks:

- The social optimum for the prisoner's dilemma is when both players cooperate – the corresponding cost sum is 2.

Definition 10.4 (dominant)**.** *A strategy is dominant if a player is never worse off by playing this strategy. A dominant strategy profile is a strategy profile in which each player plays a dominant strategy.*

Remarks:

- The dominant strategy profile in the prisoner's dilemma is when both players defect – the corresponding cost sum is 4.

Definition 10.5 (Nash Equilibrium)**.** *A Nash Equilibrium (NE) is a strategy profile in which no player can improve by unilaterally (the strategies of the other players do not change) changing its strategy.*

Remarks:

- A game can have multiple Nash Equilibria.
- In the prisoner's dilemma both players defecting is the only Nash Equilibrium.
- If every player plays a dominant strategy, then this is by definition a Nash Equilibrium.
- Nash Equilibria and dominant strategy profiles are so called solution concepts. They are used to analyze a game. There are more solution concepts, e.g. correlated equilibria or best response.
- The best response is the best strategy given a belief about the strategy of the other players. In this game the best response to both strategies of the other player is to defect. If one strategy is the best response to any strategy of the other players, it is a dominant strategy.
- If two players play the prisoner's dilemma repeatedly, it is called iterated prisoner's dilemma. It is a dominant strategy to always defect. To see this, consider the final game. Defecting is a dominant strategy. Thus, it is fixed what both players do in the last game. Now the penultimate game is the last game and by induction always defecting is a dominant strategy.
- Game theorists were invited to come up with a strategy for 200 iterations of the prisoner's dilemma to compete in a tournament. Each strategy had to play against every other strategy and accumulated points throughout the tournament. The simple Tit4Tat strategy (cooperate in the first game, then copy whatever the other player did in the previous game) won. One year later, after analyzing each strategy, another tournament (with new strategies) was held. Tit4Tat won again.
- We now look at a distributed system game.

10.3 Selfish Caching

Computers in a network want to access a file regularly. Each node $v \in V$, with V being the set of nodes and $n = |V|$, has a demand d_v for the file and wants to minimize the cost for accessing it. In order to access the file, node v can either cache the file locally which costs 1 or request the file from another node u which costs $c_{v \leftarrow u}$. If a node does not cache the file, the cost it incurs is the minimal cost to access the file remotely. Note that if no node caches the file, then every node incurs cost ∞. There is an example in Figure 10.6.

Remarks:

- We will sometimes depict this game as a graph. The cost $c_{v \leftarrow u}$ for node v to access the file from node u is equivalent to the length of the shortest path times the demand d_v.

- Note that in undirected graphs $c_{u\leftarrow v} > c_{v\leftarrow u}$ if and only if $d_u > d_v$. We assume that the graphs are undirected for the rest of the chapter.

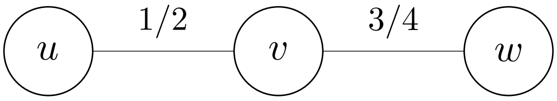


Figure 10.6: In this example we assume $d_u = d_v = d_w = 1$. Either the nodes u and w cache the file. Then neither of the three nodes has an incentive to change its behavior. The costs are 1, 1/2, and 1 for the nodes u, v, w, respectively. Alternatively, only node v caches the file. Again, neither of the three nodes has an incentive to change its behavior. The costs are 1/2, 1, and 3/4 for the nodes u, v, w, respectively.

Algorithm 10.7 Nash Equilibrium for Selfish Caching

```
$S = \{\}$                                   //set of nodes that cache the file
repeat
    Let $v$ be a node with maximum demand $d_v$ in set $V$
    $S = S \cup \{v\}, V = V \setminus \{v\}$
    Remove every node $u$ from $V$ with $c_{u\leftarrow v} \leq 1$
until V = {}
```

Theorem 10.8. *Algorithm 10.7 computes a Nash Equilibrium for Selfish Caching.*

Proof. Let u be a node that is not caching the file. Then there exists a node v for which $c_{u\leftarrow v} \leq 1$. Hence, node u has no incentive to cache.

Let u be a node that is caching the file. We now consider any other node v that is also caching the file. First, we consider the case where v cached the file before u did. Then it holds that $c_{u\leftarrow v} > 1$ by construction.

It could also be that v started caching the file after u did. Then it holds that $d_u \geq d_v$ and therefore $c_{u\leftarrow v} \geq c_{v\leftarrow u}$. Furthermore, we have $c_{v\leftarrow u} > 1$ by construction. Combining these implies that $c_{u\leftarrow v} \geq c_{v\leftarrow u} > 1$.

In either case, node u has no incentive to stop caching. □

Definition 10.9 (Price of Anarchy)**.** *Let NE_- denote the Nash Equilibrium with the highest cost (smallest payoff). The **Price of Anarchy** (PoA) is defined as*

$$PoA = \frac{\text{cost}(NE_-)}{\text{cost}(SO)}.$$

Definition 10.10 (Optimistic Price of Anarchy)**.** *Let NE_+ denote the Nash Equilibrium with the smallest cost (highest payoff). The **Optimistic Price of Anarchy** (OPoA) is defined as*

$$OPoA = \frac{\text{cost}(NE_+)}{\text{cost}(SO)}.$$

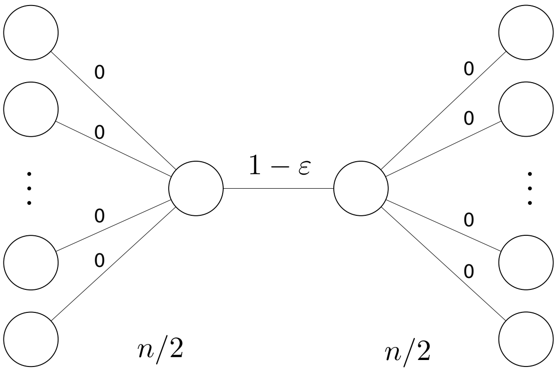


Figure 10.12: A network with a Price of Anarchy of $\Theta(n)$.

Remarks:

- The Price of Anarchy measures how much a distributed system degrades because of selfish nodes.
- We have $PoA \geq OPoA \geq 1$.

Theorem 10.11. *The (Optimistic) Price of Anarchy of Selfish Caching can be $\Theta(n)$.*

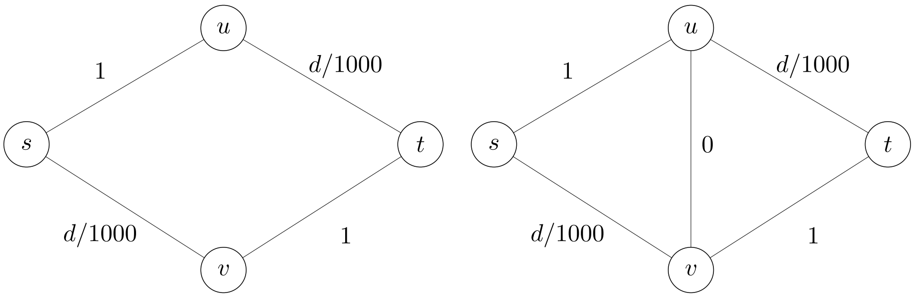
Proof. Consider a network as depicted in Figure 10.12. Every node v has demand $d_v = 1$. Note that if any node caches the file, no other node has an incentive to cache the file as well since the cost to access the file is at most $1-\varepsilon$. Without loss of generality, let us assume that a node v on the left caches the file, then it is cheaper for every node on the right to access the file remotely. Hence, the total cost of this solution is $1 + \frac{n}{2} \cdot (1-\varepsilon)$. In the social optimum one node from the left and one node from the right cache the file. This reduces the cost to 2. Hence, the Price of Anarchy is $\frac{1+\frac{n}{2}\cdot(1-\varepsilon)}{2} \underset{\varepsilon\to 0}{=} \frac{1}{2} + \frac{n}{4} = \Theta(n)$. □

10.4 Braess' Paradox

Consider the graph in Figure 10.13, it models a road network. Let us assume that there are 1000 drivers (each in their own car) that want to travel from node s to node t. Traveling along the road from s to u (or v to t) always takes 1 hour. The travel time from s to v (or u to t) depends on the traffic and increases by 1/1000 of an hour per car, i.e., when there are 500 cars driving, it takes 30 minutes to use this road.

Lemma 10.14. *Adding a super fast road (delay is 0) between u and v can increase the travel time from s to t.*

Proof. Since the drivers act rationally, they want to minimize the travel time. In the Nash Equilibrium, 500 drivers first drive to node u and then to t and 500 drivers first to node v and then to t. The travel time for each driver is 1 + 500 / 1000 = 1.5.



(a) The road network without the shortcut (b) The road network with the shortcut

Figure 10.13: Braess' Paradox, where d denotes the number of drivers using an edge.

To reduce congestion, a super fast road (delay is 0) is built between nodes u and v. This results in the following Nash Equilibrium: every driver now drives from s to v to u to t. The total cost is now $2 > 1.5$. □

Remarks:

- There are physical systems which exhibit similar properties. Some famous ones employ a spring. YouTube has some fascinating videos about this. Simply search for "Braess Paradox Spring".
- We will now look at another famous game that will allow us to deepen our understanding of game theory.

10.5 Rock-Paper-Scissors

There are two players, u and v. Each player simultaneously chooses one of three options: rock, paper, or scissors. The rules are simple: paper beats rock, rock beats scissors, and scissors beat paper. A matrix representation of this game is in Table 10.15.

u v		Player u		
		Rock	Paper	Scissors
Player v	Rock	0 0	1 -1	-1 1
	Paper	-1 1	0 0	1 -1
	Scissors	1 -1	-1 1	0 0

Table 10.15: Rock-Paper-Scissors as a matrix.

Remarks:

- None of the three strategies is a Nash Equilibrium. Whatever player u chooses, player v can always switch her strategy such that she wins.
- This is highlighted in the best response concept. The best response to e.g. scissors is to play rock. The other player switches to paper. And so on.
- Is this a game without a Nash Equilibrium? John Nash answered this question in 1950. By choosing each strategy with a certain probability, we can obtain a so called Mixed Nash Equilibrium.

Definition 10.16 (Mixed Nash Equilibrium)**.** *A Mixed Nash Equilibrium (MNE) is a strategy profile in which at least one player is playing a randomized strategy (choose strategy profiles according to probabilities), and no player can improve their expected payoff by unilaterally changing their (randomized) strategy.*

Theorem 10.17. *Every game has a mixed Nash Equilibrium.*

Remarks:

- The Nash Equilibrium of this game is if both players choose each strategy with probability 1/3. The expected payoff is 0.
- Any strategy (or mix of them) is a best response to a player choosing each strategy with probability 1/3.
- In a pure Nash Equilibrium, the strategies are chosen deterministically. Rock-Paper-Scissors does not have a pure Nash Equilibrium.
- Even though every game has a mixed Nash Equilibrium. Sometimes such an equilibrium is computationally difficult to compute. One should be cautious about economic assumptions such as “the market will always find the equilibrium”.
- Unfortunately, game theory does not always model problems accurately. Many real world problems are too complex to be captured by a game. And as you may know, humans (not only politicians) are often not rational.
- In distributed systems, players can be servers, routers, etc. Game theory can tell us whether systems and protocols are prone to selfish behavior.

10.6 Mechanism Design

Whereas game theory analyzes existing systems, there is a related area that focuses on designing games – mechanism design. The task is to create a game where nodes have an incentive to behave “nicely”.

Definition 10.18 (auction)**.** *One good is sold to a group of bidders in an auction. Each bidder v_i has a secret value z_i for the good and tells his bid b_i to the auctioneer. The auctioneer sells the good to one bidder for a price p.*

Remarks:

- For simplicity, we assume that no two bids are the same, and that $b_1 > b_2 > b_3 > \ldots$

Algorithm 10.19 First Price Auction

```
every bidder $v_i$ submits his bid $b_i$
the good is allocated to the highest bidder $v_1$ for the price $p = b_1$
```

Definition 10.20 (truthful)**.** *An auction is truthful if no player v_i can gain anything by not stating the truth, i.e., $b_i = z_i$.*

Theorem 10.21. *A First Price Auction (Algorithm 10.19) is not truthful.*

Proof. Consider an auction with two bidders, with bids b_1 and b_2. By not stating the truth and decreasing his bid to $b_1 - \varepsilon > b_2$, player one could pay less and thus gain more. Thus, the first price auction is not truthful. □

Algorithm 10.22 Second Price Auction

```
every bidder $v_i$ submits his bid $b_i$
the good is allocated to the highest bidder $v_1$ for $p = b_2$
```

Theorem 10.23. *Truthful bidding is a dominant strategy in a Second Price Auction.*

Proof. Let z_i be the truthful value of node v_i and b_i his bid. Let $b_{\max} = \max_{j \neq i} b_j$ is the largest bid from other nodes but v_i. The payoff for node v_i is $z_i - b_{\max}$ if $b_i > b_{\max}$ and 0 else. Let us consider overbidding first, i.e., $b_i > z_i$:

- If $b_{\max} < z_i < b_i$, then both strategies win and yield the same payoff $(z_i - b_{\max})$.
- If $z_i < b_i < b_{\max}$, then both strategies lose and yield a payoff of 0.
- If $z_i < b_{\max} < b_i$, then overbidding wins the auction, but the payoff $(z_i - b_{\max})$ is negative. Truthful bidding loses and yields a payoff of 0.

Likewise underbidding, i.e. $b_i < z_i$:

- If $b_{\max} < b_i < z_i$, then both strategies win and yield the same payoff $(z_i - b_{\max})$.
- If $b_i < z_i < b_{\max}$, then both strategies lose and yield a payoff of 0.
- If $b_i < b_{\max} < z_i$, then truthful bidding wins and yields a positive payoff $(z_i - b_{\max})$. Underbidding loses and yields a payoff of 0.

Hence, truthful bidding is a dominant strategy for each node v_i. □

Remarks:

- Let us use this for Selfish Caching. We need to choose a node that is the first to cache the file. But how? By holding an auction. Every node says for which price it is willing to cache the file. We pay the node with the lowest offer and pay it the second lowest offer to ensure truthful offers.
- Since a mechanism designer can manipulate incentives, she can implement a strategy profile by making all the strategies in this profile dominant.

Theorem 10.24. *Any Nash Equilibrium of Selfish Caching can be implemented for free.*

Proof. If the mechanism designer wants the nodes from the caching set S of the Nash Equilibrium to cache, then she can offer the following deal to every node not in S: "If any node from set S does not cache the file, then I will ensure a positive payoff for you." Thus, all nodes not in S prefer not to cache since this is a dominant strategy for them. Consider now a node $v \in S$. Since S is a Nash Equilibrium, node v incurs cost of at least 1 if it does not cache the file. For nodes that incur cost of exactly 1, the mechanism designer can even issue a penalty if the node does not cache the file. Thus, every node $v \in S$ caches the file. □

Remarks:

- Mechanism design assumes that the players act rationally and want to maximize their payoff. In real-world distributed systems some players may be not selfish, but actively malicious (byzantine).

- What about P2P file sharing? To increase the overall experience, BitTorrent suggests that peers offer better upload speed to peers who upload more. This idea can be exploited. By always claiming to have nothing to trade yet, the BitThief client downloads without uploading. In addition to that, it connects to more peers than the standard client to increase its download speed.

- Many techniques have been proposed to limit such free riding behavior, e.g., tit-for-tat trading: I will only share something with you if you share something with me. To solve the bootstrap problem ("I don't have anything yet"), nodes receive files or pieces of files whose hash match their own hash for free. One can also imagine indirect trading. Peer u uploads to peer v, who uploads to peer w, who uploads to peer u. Finally, one could imagine using virtual currencies or a reputation system (a history of who uploaded what). Reputation systems suffer from collusion and Sybil attacks. If one node pretends to be many nodes who rate each other well, it will have a good reputation.

Chapter Notes

Game theory was started by a proof for mixed-strategy equilibria in two-person zero-sum games by John von Neumann [Neu28]. Later, von Neumann and Morgenstern introduced game theory to a wider audience [NM44]. In 1950 John Nash proved that every game has a mixed Nash Equilibrium [Nas50]. The Prisoner's Dilemma was first formalized by Flood and Dresher [Flo52]. The iterated prisoner's dilemma tournament was organized by Robert Axelrod [AH81]. The Price of Anarchy definition is from Koutsoupias and Papadimitriou [KP99]. This allowed the creation of the Selfish Caching Game [CCW$^+$04], which we used as a running example in this chapter. Braess' paradox was discovered by Dietrich Braess in 1968 [Bra68]. A generalized version of the second-price auction is the VCG auction, named after three successive papers from first Vickrey, then Clarke, and finally Groves [Vic61, Cla71, Gro73]. One popular example of selfishness in practice is BitThief – a BitTorrent client that successfully downloads without uploading [LMSW06]. Using game theory economists try to understand markets and predict crashes. Apart from John Nash, the Sveriges Riksbank Prize (Nobel Prize) in Economics has been awarded many times to game theorists. For example in 2007 Hurwicz, Maskin, and Myerson received the prize for "for having laid the foundations of mechanism design theory". There is a considerable amount of work on mixed adversarial models with byzantine, altruistic, and rational ("BAR") players, e.g., [AAC$^+$05, ADGH06, MSW06]. Daskalakis et al. [DGP09] showed that computing a Nash Equilibrium may not be trivial.

This chapter was written in collaboration with Philipp Brandes.

Bibliography

[AAC$^+$05] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Michael Dahlin, Jean-Philippe Martin, and Carl Porth. BAR fault tolerance for cooperative services. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*, pages 45–58, 2005.

[ADGH06] Ittai Abraham, Danny Dolev, Rica Gonen, and Joseph Y. Halpern. Distributed computing meets game theory: robust mechanisms for rational secret sharing and multiparty computation. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing, PODC 2006, Denver, CO, USA, July 23-26, 2006*, pages 53–62, 2006.

[AH81] Robert Axelrod and William Donald Hamilton. The evolution of cooperation. *Science*, 211(4489):1390–1396, 1981.

[Bra68] Dietrich Braess. Über ein paradoxon aus der verkehrsplanung. *Unternehmensforschung*, 12(1):258–268, 1968.

[CCW$^+$04] Byung-Gon Chun, Kamalika Chaudhuri, Hoeteck Wee, Marco Barreno, Christos H Papadimitriou, and John Kubiatowicz. Selfish caching in distributed systems: a game-theoretic analysis. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 21–30. ACM, 2004.

[Cla71] Edward H Clarke. Multipart pricing of public goods. *Public choice*, 11(1):17–33, 1971.

[DGP09] Constantinos Daskalakis, Paul W. Goldberg, and Christos H. Papadimitriou. The complexity of computing a nash equilibrium. *SIAM J. Comput.*, 39(1):195–259, 2009.

[Flo52] Merrill M Flood. Some experimental games. *Management Science*, 5(1):5–26, 1952.

[Gro73] Theodore Groves. Incentives in teams. *Econometrica: Journal of the Econometric Society*, pages 617–631, 1973.

[KP99] Elias Koutsoupias and Christos Papadimitriou. Worst-case equilibria. In *STACS 99*, pages 404–413. Springer, 1999.

[LMSW06] Thomas Locher, Patrick Moor, Stefan Schmid, and Roger Wattenhofer. Free Riding in BitTorrent is Cheap. In *5th Workshop on Hot Topics in Networks (HotNets), Irvine, California, USA*, November 2006.

[MSW06] Thomas Moscibroda, Stefan Schmid, and Roger Wattenhofer. When selfish meets evil: byzantine players in a virus inoculation game. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing, PODC 2006, Denver, CO, USA, July 23-26, 2006*, pages 35–44, 2006.

[Nas50] John F. Nash. Equilibrium points in n-person games. *Proc. Nat. Acad. Sci. USA*, 36(1):48–49, 1950.

[Neu28] John von Neumann. Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen*, 100(1):295–320, 1928.

[NM44] John von Neumann and Oskar Morgenstern. *Theory of games and economic behavior*. Princeton university press, 1944.

[Vic61] William Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *The Journal of finance*, 16(1):8–37, 1961.

Chapter 11

Distributed Storage

How do you store 1M movies, each with a size of about 1GB, on 1M nodes, each equipped with a 1TB disk? Simply store the movies on the nodes, arbitrarily, and memorize (with a global index) which movie is stored on which node. What if the set of movies or nodes changes over time, and you do not want to change your global index too often?

11.1 Consistent Hashing

Several variants of hashing will do the job, e.g. consistent hashing:

Algorithm 11.1 Consistent Hashing

```
Hash the unique file name of each movie $x$ with a known set of hash functions
    $h_i(x) \rightarrow [0,1)$, for $i = 1, \dots, k$
Hash the unique name (e.g., IP address and port number) of each node with
    the same hash function $h(u) \rightarrow [0,1)$
Store a copy of movie $x$ on node $u$ if $h_i(x) \approx h(u)$, for any $i$. More formally,
    store movie $x$ on node $u$ if
        $|h_i(x) - h(u)| = \min_v\{|h_i(x) - h(v)|\}$, for any $i$
```

Theorem 11.2 (Consistent Hashing)**.** *In expectation, each node in Algorithm 11.1 stores km/n movies, where k is the number of hash functions, m the number of different movies and n the number of nodes.*

Proof. For a specific movie (out of m) and a specific hash function (out of k), all n nodes have the same probability $1/n$ to hash closest to the movie hash. By linearity of expectation, each node stores km/n movies in expectation if we also count duplicates of movies on a node. □

Remarks:

- Let us do a back-of-the-envelope calculation. We have $m = 1\text{M}$ movies, $n = 1\text{M}$ nodes, each node has storage for 1TB/1GB = 1K movies, i.e., we use $k = 1\text{K}$ hash functions. Theorem 11.2 shows each node stores about 1K movies.

- Using the Chernoff bound below with $\mu = km/n = 1\text{K}$, the probability that a node uses 10% more memory than expected is less than 1%.

Facts 11.3. *A version of a* ***Chernoff bound*** *states the following:*
Let $x_1, \ldots, x_n$ be independent Bernoulli-distributed random variables with $Pr[x_i = 1] = p_i$ and $Pr[x_i = 0] = 1 - p_i = q_i$, then for $X := \sum_{i=1}^{n} x_i$ and $\mu := \mathbb{E}[X] = \sum_{i=1}^{n} p_i$ the following holds:

$$\textit{for any } \delta > 0\textit{: } Pr[X \geq (1+\delta)\mu] < \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{\mu}$$

Remarks:

- Instead of storing movies directly on nodes as in Algorithm 11.1, we can also store the movies on any nodes we like. The nodes of Algorithm 11.1 then simply store forward pointers to the actual movie locations.

- For better load balancing, we might also hash nodes multiple times.

- In this chapter we want to push unreliability to the extreme. What if the nodes are so unreliable that on average a node is only available for 1 hour? In other words, nodes exhibit a high *churn*, they constantly join and leave the distributed system.

- With such a high churn, hundreds or thousands of nodes will change every second. No single node can have an accurate picture of what other nodes are currently in the system. This is remarkably different to classic distributed systems, where a single unavailable node may already be a minor disaster: all the other nodes have to get a consistent view (Definition 13.5) of the system again. In high churn systems it is impossible to have a consistent view at any time.

- Instead, each node will just know about a small subset of 100 or less other nodes (“neighbors”). This way, nodes can withstand high churn situations.

- On the downside, nodes will not directly know which node is responsible for what movie. Instead, a node searching for a movie might have to ask a neighbor node, which in turn will recursively ask another neighbor node, until the correct node storing the movie (or a forward pointer to the movie) is found. The nodes of our distributed storage system form a virtual network, also called an *overlay network*.

11.2 Hypercubic Networks

In this section we present a few overlay topologies of general interest.

Definition 11.4 (Topology Properties)**.** *Our virtual network should have the following properties:*

- *The network should be (somewhat)* ***homogeneous****: no node should play a dominant role, no node should be a single point of failure.*
- *The nodes should have* ***IDs****, and the IDs should span the universe* $[0, 1)$*, such that we can store data with hashing, as in Algorithm 11.1.*
- *Every node should have a small* ***degree****, if possible polylogarithmic in* n*, the number of nodes. This will allow every node to maintain a persistent connection with each neighbor, which will help us to deal with churn.*
- *The network should have a small* ***diameter****, and routing should be easy. If a node does not have the information about a data item, then it should know which neighbor to ask. Within a few (polylogarithmic in* n*) hops, one should find the node that has the correct information.*

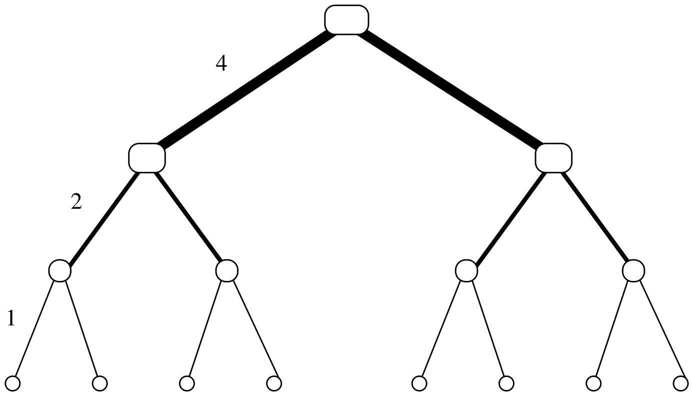


Figure 11.5: The structure of a fat tree.

Remarks:

- Some basic network topologies used in practice are trees, rings, grids or tori. Many other suggested networks are simply combinations or derivatives of these.
- The advantage of trees is that the routing is very easy: for every source-destination pair there is only one path. However, since the root of a tree is a bottleneck, trees are not homogeneous. Instead, so-called *fat trees* should be used. Fat trees have the property that every edge connecting a node v to its parent u has a capacity that is proportional to the number of leaves of the subtree rooted at v. See Figure 11.5 for a picture.

- Fat trees belong to a family of networks that require edges of non-uniform capacity to be efficient. Networks with edges of uniform capacity are easier to build. This is usually the case for grids and tori. Unless explicitly mentioned, we will treat all edges in the following to be of capacity 1.

Definition 11.6 (Torus, Mesh)**.** *Let $m, d \in \mathbb{N}$. The (m, d)-**mesh** $M(m, d)$ is a graph with node set $V = [m]^d$ and edge set*

$$E = \left\{ \{(a_1, \ldots, a_d), (b_1, \ldots, b_d)\} \mid a_i, b_i \in [m], \sum_{i=1}^{d} |a_i - b_i| = 1 \right\},$$

*where $[m]$ means the set $\{0, \ldots, m-1\}$. The (m, d)-**torus** $T(m, d)$ is a graph that consists of an (m, d)-mesh and additionally wrap-around edges from nodes $(a_1, \ldots, a_{i-1}, m-1, a_{i+1}, \ldots, a_d)$ to nodes $(a_1, \ldots, a_{i-1}, 0, a_{i+1}, \ldots, a_d)$ for all $i \in \{1, \ldots, d\}$ and all $a_j \in [m]$ with $j \neq i$. In other words, we take the expression $a_i - b_i$ in the sum modulo m prior to computing the absolute value. $M(m, 1)$ is also called a **path**, $T(m, 1)$ a **cycle**, and $M(2, d) = T(2, d)$ a d-**dimensional hypercube**. Figure 11.7 presents a linear array, a torus, and a hypercube.*

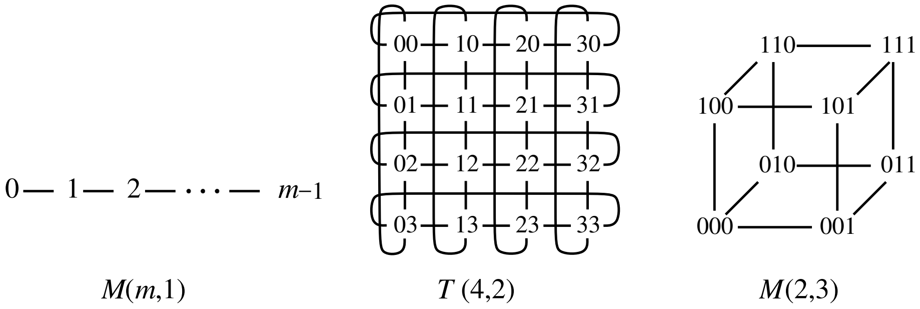


Figure 11.7: The structure of $M(m, 1)$, $T(4, 2)$, and $M(2, 3)$.

Remarks:

- Routing on a mesh, torus, or hypercube is trivial. On a d-dimensional hypercube, to get from a source bitstring s to a target bitstring t one only needs to fix each "wrong" bit, one at a time; in other words, if the source and the target differ by k bits, there are $k!$ routes with k hops.

- As required by Definition 11.4, the d-bit IDs of the nodes need to be mapped to the universe $[0, 1)$. One way to do this is by interpreting an ID as the binary representation of the fractional part of a decimal number. For example, the ID **101** is mapped to $\mathbf{0.101_2}$ which has a decimal value of $\mathbf{0} \cdot 2^0 + \mathbf{1} \cdot 2^{-1} + \mathbf{0} \cdot 2^{-2} + \mathbf{1} \cdot 2^{-3} = \frac{5}{8}$.

- The Chord architecture is a close relative of the hypercube, basically a less rigid hypercube. The hypercube connects every node with an ID in $[0, 1)$ with every node in *exactly* distance 2^{-i}, $i = 1, 2, \ldots, d$ in $[0, 1)$. Chord instead connect nodes with *approximately* distance 2^{-i}.

- The hypercube has many derivatives, the so-called *hypercubic networks*. Among these are the butterfly, cube-connected-cycles, shuffle-exchange, and de Bruijn graph. We start with the butterfly, which is basically a "rolled out" hypercube.

Definition 11.8 (Butterfly)**.** *Let $d \in \mathbb{N}$. The **d-dimensional butterfly** $BF(d)$ is a graph with node set $V = [d+1] \times [2]^d$ and an edge set $E = E_1 \cup E_2$ with*

$$E_1 = \{\{(i,\alpha),(i+1,\alpha)\} \mid i \in [d],\ \alpha \in [2]^d\}$$

and

$$E_2 \quad = \quad \{\{(i,\alpha),(i+1,\beta)\} \mid i \in [d],\ \alpha,\beta \in [2]^d, \alpha \oplus \beta = 2^i\}.$$

*A node set $\{(i,\alpha) \mid \alpha \in [2]^d\}$ is said to form **level** i of the butterfly. The **d-dimensional wrap-around butterfly** W-BF(d) is defined by taking the $BF(d)$ and having $(d,\alpha) = (0,\alpha)$ for all $\alpha \in [2]^d$.*

Remarks:

- Figure 11.9 shows the 3-dimensional butterfly $BF(3)$. The $BF(d)$ has $(d+1)2^d$ nodes, $2d \cdot 2^d$ edges and degree 4. It is not difficult to check that combining the node sets $\{(i,\alpha) \mid i \in [d]\}$ for all $\alpha \in [2]^d$ into a single node results in the hypercube.

- Butterflies have the advantage of a constant node degree over hypercubes, whereas hypercubes feature more fault-tolerant routing.

- You may have seen butterfly-like structures before, e.g. sorting networks, communication switches, data center networks, fast fourier transform (FFT). The Benes network (telecommunication) is nothing but two back-to-back butterflies. The Clos network (data centers) is a close relative to Butterflies too. Actually, merging the 2^i nodes on level i that share the first $d-i$ bits into a single node, the Butterfly becomes a fat tree. Every year there are new applications for which hypercubic networks are the perfect solution!

- Next we define the cube-connected-cycles network. It only has a degree of 3 and it results from the hypercube by replacing the corners by cycles.

Definition 11.10 (Cube-Connected-Cycles)**.** *Let $d \in \mathbb{N}$. The **cube-connected-cycles** network CCC(d) is a graph with node set $V = \{(a,p) \mid a \in [2]^d, p \in [d]\}$ and edge set*

$$\begin{aligned} E \quad = \quad & \big\{\{(a,p),(a,(p+1) \bmod d)\} \mid a \in [2]^d, p \in [d]\big\} \\ & \cup \big\{\{(a,p),(b,p)\} \mid a,b \in [2]^d, p \in [d], |a-b| = 2^p\big\} \end{aligned}$$

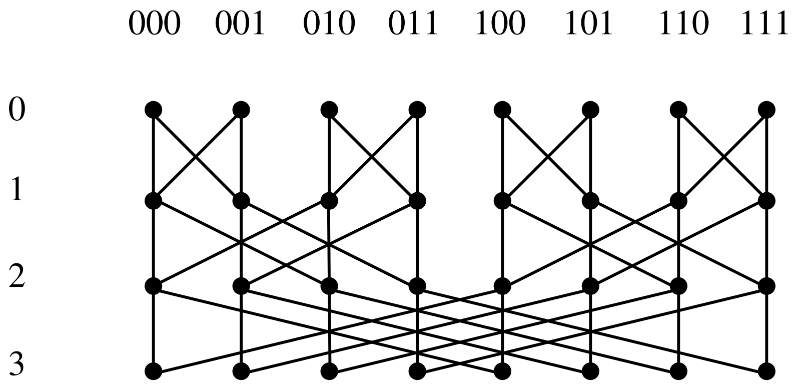


Figure 11.9: The structure of BF(3).

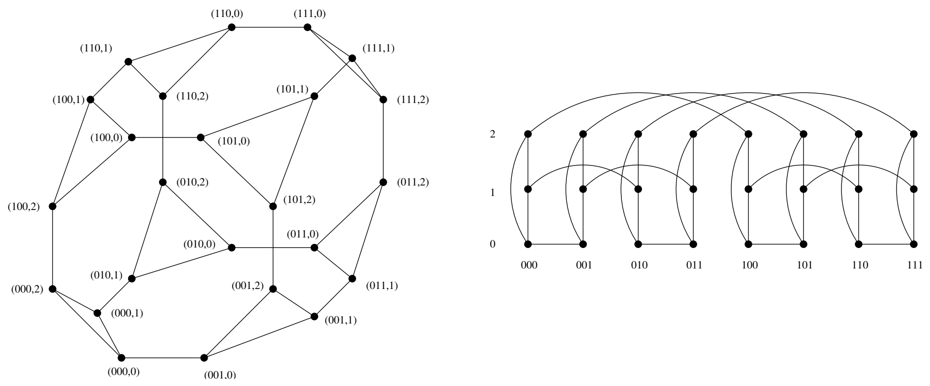


Figure 11.11: The structure of CCC(3).

Remarks:

- Two possible representations of a CCC can be found in Figure 11.11.
- The shuffle-exchange is yet another way of transforming the hypercubic interconnection structure into a constant degree network.

Definition 11.12 (Shuffle-Exchange)**.** *Let $d \in \mathbb{N}$. The **d-dimensional shuffle-exchange** $SE(d)$ is defined as an undirected graph with node set $V = [2]^d$ and an edge set $E = E_1 \cup E_2$ with*

$$E_1 = \{\{(a_1, \ldots, a_d), (a_1, \ldots, \bar{a}_d)\} \mid (a_1, \ldots, a_d) \in [2]^d,\ \bar{a}_d = 1 - a_d\}$$

and

$$E_2 = \{\{(a_1, \ldots, a_d), (a_d, a_1, \ldots, a_{d-1})\} \mid (a_1, \ldots, a_d) \in [2]^d\}\ .$$

Figure 11.13 shows the 3- and 4-dimensional shuffle-exchange graph.

Definition 11.14 (DeBruijn)**.** *The **b-ary DeBruijn graph of dimension** d $DB(b, d)$ is an undirected graph $G = (V, E)$ with node set $V = \{v \in [b]^d\}$*

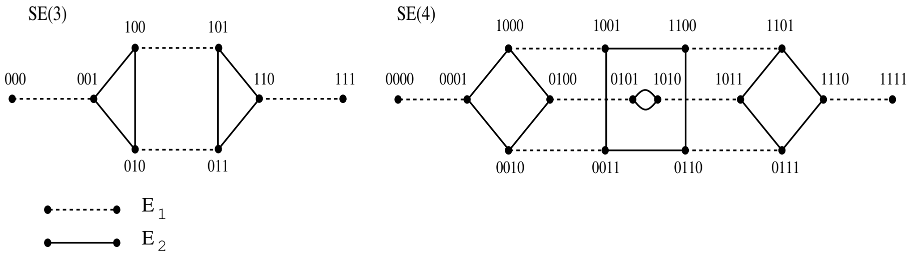


Figure 11.13: The structure of SE(3) and SE(4).

and edge set E that contains all edges $\{v, w\}$ with the property that $w \in \{(x, v_1, \ldots, v_{d-1}) : x \in [b]\}$, where $v = (v_1, \ldots, v_d)$.

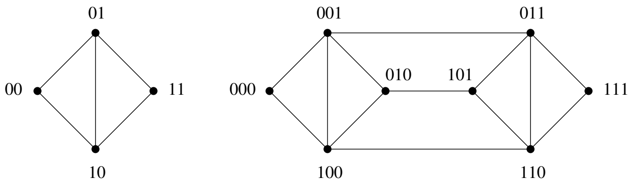


Figure 11.15: The structure of $DB(2, 2)$ and $DB(2, 3)$.

Remarks:

- Two examples of a DeBruijn graph can be found in Figure 11.15.
- There are some data structures which also qualify as hypercubic networks. An example of a hypercubic network is the skip list, the balanced binary search tree for the lazy programmer:

Definition 11.16 (Skip List)**.** *The skip list is an ordinary ordered linked list of objects, augmented with additional forward links. The ordinary linked list is the level 0 of the skip list. In addition, every object is promoted to level 1 with probability 1/2. As for level 0, all level 1 objects are connected by a linked list. In general, every object on level i is promoted to the next level with probability 1/2. A special start-object points to the smallest/first object on each level.*

Remarks:

- Search, insert, and delete can be implemented in $\mathcal{O}(\log n)$ expected time in a skip list, simply by jumping from higher levels to lower ones when overshooting the searched position. Also, the amortized memory cost of each object is constant, as on average an object only has two forward links.
- The randomization can easily be discarded, by deterministically promoting a constant fraction of objects of level i to level $i + 1$, for all

i. When inserting or deleting, object o simply checks whether its left and right level i neighbors are being promoted to level $i+1$. If none of them is, promote object o itself. Essentially we establish a maximal independent set (MIS) on each level, hence at least every third and at most every second object is promoted.

- There are obvious variants of the skip list, e.g., the skip graph. Instead of promoting only half of the nodes to the next level, we always promote all the nodes, similarly to a balanced binary tree: All nodes are part of the root level of the binary tree. Half the nodes are promoted left, and half the nodes are promoted right, on each level. Hence on level i we have have 2^i lists (or, if we connect the last element again with the first: rings) of about $n/2^i$ objects. The skip graph features all the properties of Definition 11.4.

- More generally, how are degree and diameter of Definition 11.4 related? The following theorem gives a general lower bound.

Theorem 11.17. *Every graph of maximum degree $d > 2$ and size n must have a diameter of at least $\lceil(\log n)/(\log(d-1))\rceil - 2$.*

Proof. Suppose we have a graph $G = (V, E)$ of maximum degree d and size n. Start from any node $v \in V$. In a first step at most d other nodes can be reached. In two steps at most $d \cdot (d-1)$ additional nodes can be reached. Thus, in general, in at most r steps at most

$$1 + \sum_{i=0}^{r-1} d \cdot (d-1)^i = 1 + d \cdot \frac{(d-1)^r - 1}{(d-1) - 1} \leq \frac{d \cdot (d-1)^r}{d-2}$$

nodes (including v) can be reached. This has to be at least n to ensure that v can reach all other nodes in V within r steps. Hence,

$$(d-1)^r \geq \frac{(d-2) \cdot n}{d} \quad \Leftrightarrow \quad r \geq \log_{d-1}((d-2) \cdot n/d) .$$

Since $\log_{d-1}((d-2)/d) > -2$ for all $d > 2$, this is true only if $r \geq \lceil(\log n)/(\log(d-1))\rceil - 2$. □

Remarks:

- In other words, constant-degree hypercubic networks feature an asymptotically optimal diameter D.

- Other hypercubic graphs manage to have a different tradeoff between node degree d and diameter D. The pancake graph, for instance, minimizes the maximum of these with $\max(d, D) = \Theta(\log n/ \log\log n)$. The ID of a node u in the pancake graph of dimension d is an arbitrary permutation of the numbers $1, 2, \ldots, d$. Two nodes u, v are connected by an edge if one can get the ID of node v by taking the ID of node u, and reversing (flipping) the first k (for $k = 1, \ldots, d$) numbers of u's ID. For example, in dimension $d = 4$, nodes $u = 2314$ and $v = 1324$ are neighbors.

- There are a few other interesting graph classes which are not hypercubic networks, but nevertheless seem to relate to the properties of Definition 11.4. Small-world graphs (a popular representations for social networks) also have small diameter, however, in contrast to hypercubic networks, they are not homogeneous and feature nodes with large degrees.

- Expander graphs (an expander graph is a sparse graph which has good connectivity properties, that is, from every not too large subset of nodes you are connected to an even larger set of nodes) are homogeneous, have a low degree and small diameter. However, expanders are often not routable.

11.3 DHT & Churn

Definition 11.18 (Distributed Hash Table (DHT))**.** *A* ***distributed hash table*** *(DHT) is a distributed data structure that implements a distributed storage. A DHT should support at least (i) a search (for a key) and (ii) an insert (key, object) operation, possibly also (iii) a delete (key) operation.*

Remarks:

- A DHT has many applications beyond storing movies, e.g., the Internet domain name system (DNS) is essentially a DHT.

- A DHT can be implemented as a hypercubic overlay network with nodes having identifiers such that they span the ID space $[0, 1)$.

- A hypercube can directly be used for a DHT. Just use a globally known set of hash functions h_i, mapping movies to bit strings with d bits.

- Other hypercubic structures may be a bit more intricate when using it as a DHT: The butterfly network, for instance, may directly use the $d+1$ layers for replication, i.e., all the $d+1$ nodes are responsible for the same ID.

- Other hypercubic networks, e.g. the pancake graph, might need a bit of twisting to find appropriate IDs.

- We assume that a joining node knows a node which already belongs to the system. This is known as the bootstrap problem. Typical solutions are: If a node has been connected with the DHT previously, just try some of these previous nodes. Or the node may ask some authority for a list of IP addresses (and ports) of nodes that are regularly part of the DHT.

- Many DHTs in the literature are analyzed against an adversary that can crash a fraction of random nodes. After crashing a few nodes the system is given sufficient time to recover again. However, this seems unrealistic. The scheme sketched in this section significantly differs from this in two major aspects.

- First, we assume that joins and leaves occur in a worst-case manner. We think of an adversary that can remove and add a bounded number of nodes; the adversary can choose which nodes to crash and how nodes join.

- Second, the adversary does not have to wait until the system is recovered before it crashes the next batch of nodes. Instead, the adversary can constantly crash nodes, while the system is trying to stay alive. Indeed, the system is *never fully repaired* but *always fully functional.* In particular, the system is resilient against an adversary that continuously attacks the "weakest part" of the system. The adversary could for example insert a crawler into the DHT, learn the topology of the system, and then repeatedly crash selected nodes, in an attempt to partition the DHT. The system counters such an adversary by continuously moving the remaining or newly joining nodes towards the areas under attack.

- Clearly, we cannot allow the adversary to have unbounded capabilities. In particular, in any constant time interval, the adversary can at most add and/or remove $O(\log n)$ nodes, n being the total number of nodes currently in the system. This model covers an adversary which repeatedly takes down nodes by a distributed denial of service attack, however only a logarithmic number of nodes at each point in time. The algorithm relies on messages being delivered timely, in at most constant time between any pair of operational nodes, i.e., the synchronous model. Using the trivial synchronizer this is not a problem. We only need bounded message delays in order to have a notion of time which is needed for the adversarial model. The duration of a round is then proportional to the propagation delay of the slowest message.

Algorithm 11.19 DHT

```
Given: a globally known set of hash functions $h_i$, and a hypercube (or any other hypercubic network)
Each hypercube virtual node ("hypernode") consists of $\Theta(\log n)$ nodes.
Nodes have connections to all other nodes of their hypernode and to nodes of their neighboring hypernodes.
Because of churn, some of the nodes have to change to another hypernode such that up to constant factors, all hypernodes own the same number of nodes at all times.
If the total number of nodes $n$ grows or shrinks above or below a certain threshold, the dimension of the hypercube is increased or decreased by one, respectively.
```

Remarks:

- Having a logarithmic number of hypercube neighbors, each with a logarithmic number of nodes, means that each node has $\Theta(\log^2 n)$ neighbors. However, with some additional bells and whistles one can achieve $\Theta(\log n)$ neighbor nodes.

- The balancing of nodes among the hypernodes can be seen as a dynamic token distribution problem on the hypercube. Each hypernode has a certain number of tokens, the goal is to distribute the tokens along the edges of the graph such that all hypernodes end up with the same or almost the same number of tokens. While tokens are moved around, an adversary constantly inserts and deletes tokens. See also Figure 11.20.

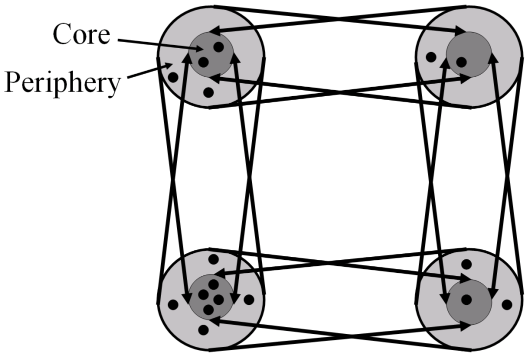


Figure 11.20: A simulated 2-dimensional hypercube with four hypernodes, each consisting of several nodes. Also, all the nodes are either in the core or in the periphery of a node. All nodes within the same hypernode are completely connected to each other, and additionally, all nodes of a hypernode are connected to the core nodes of the neighboring nodes. Only the core nodes store data items, while the peripheral nodes move between the nodes to balance biased adversarial churn.

- In summary, the storage system builds on two basic components: (i) an algorithm which performs the described dynamic token distribution and (ii) an information aggregation algorithm which is used to estimate the number of nodes in the system and to adapt the dimension of the hypercube accordingly:

Theorem 11.21 (DHT with Churn). *We have a fully scalable, efficient distributed storage system which tolerates* $\mathrm{O}(\log n)$ *worst-case joins and/or crashes per constant time interval. As in other storage systems, nodes have* $\mathrm{O}(\log n)$ *overlay neighbors, and the usual operations (e.g., search, insert) take time* $\mathrm{O}(\log n)$.

Remarks:

- Indeed, handling churn is only a minimal requirement to make a distributed storage system work. Advanced studies proposed more elaborate architectures which can also handle other security issues, e.g., privacy or Byzantine attacks.

Chapter Notes

The ideas behind distributed storage were laid during the peer-to-peer (P2P) file sharing hype around the year 2000, so a lot of the seminal research in this area is labeled P2P. The paper of Plaxton, Rajaraman, and Richa [PRR97] laid out a blueprint for many so-called structured P2P architecture proposals, such as Chord [SMK$^+$01], CAN [RFH$^+$01], Pastry [RD01], Viceroy [MNR02], Kademlia [MM02], Koorde [KK03], SkipGraph [AS03], SkipNet [HJS$^+$03], or Tapestry [ZHS$^+$04]. Also the paper of Plaxton et. al. was standing on the shoulders of giants. Some of its eminent precursors are: linear and consistent hashing [KLL$^+$97], locating shared objects [AP90, AP91], compact routing [SK85, PU88], and even earlier: hypercubic networks, e.g. [AJ75, Wit81, GS81, BA84].

Furthermore, the techniques in use for prefix-based overlay structures are related to a proposal called LAND, a locality-aware distributed hash table proposed by Abraham et al. [AMD04].

More recently, a lot of P2P research focussed on security aspects, describing for instance attacks [LMSW06, SENB07, Lar07], and provable countermeasures [KSW05, AS09, BSS09]. Another topic currently garnering interest is using P2P to help distribute live streams of video content on a large scale [LMSW07]. There are several recommendable introductory books on P2P computing, e.g. [SW05, SG05, MS07, KW08, BYL08].

Some of the figures in this chapter have been provided by Christian Scheideler.

Bibliography

[AJ75] George A. Anderson and E. Douglas Jensen. Computer Interconnection Structures: Taxonomy, Characteristics, and Examples. *ACM Comput. Surv.*, 7(4):197–213, December 1975.

[AMD04] Ittai Abraham, Dahlia Malkhi, and Oren Dobzinski. LAND: stretch (1 + epsilon) locality-aware networks for DHTs. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '04, pages 550–559, Philadelphia, PA, USA, 2004. Society for Industrial and Applied Mathematics.

[AP90] Baruch Awerbuch and David Peleg. Efficient Distributed Construction of Sparse Covers. Technical report, The Weizmann Institute of Science, 1990.

[AP91] Baruch Awerbuch and David Peleg. Concurrent Online Tracking of Mobile Users. In *SIGCOMM*, pages 221–233, 1991.

[AS03] James Aspnes and Gauri Shah. Skip Graphs. In *SODA*, pages 384–393. ACM/SIAM, 2003.

[AS09] Baruch Awerbuch and Christian Scheideler. Towards a Scalable and Robust DHT. *Theory Comput. Syst.*, 45(2):234–260, 2009.

[BA84] L. N. Bhuyan and D. P. Agrawal. Generalized Hypercube and Hyperbus Structures for a Computer Network. *IEEE Trans. Comput.*, 33(4):323–333, April 1984.

[BSS09] Matthias Baumgart, Christian Scheideler, and Stefan Schmid. A DoS-resilient information system for dynamic data management. In

Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures, SPAA '09, pages 300–309, New York, NY, USA, 2009. ACM.

[BYL08] John Buford, Heather Yu, and Eng Keong Lua. *P2P Networking and Applications.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[GS81] J.R. Goodman and C.H. Sequin. Hypertree: A Multiprocessor Interconnection Topology. *Computers, IEEE Transactions on*, C-30(12):923–933, dec. 1981.

[HJS$^+$03] Nicholas J. A. Harvey, Michael B. Jones, Stefan Saroiu, Marvin Theimer, and Alec Wolman. SkipNet: a scalable overlay network with practical locality properties. In *Proceedings of the 4th conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, pages 9–9, Berkeley, CA, USA, 2003. USENIX Association.

[KK03] M. Frans Kaashoek and David R. Karger. Koorde: A Simple Degree-Optimal Distributed Hash Table. In M. Frans Kaashoek and Ion Stoica, editors, *IPTPS*, volume 2735 of *Lecture Notes in Computer Science*, pages 98–107. Springer, 2003.

[KLL$^+$97] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In Frank Thomson Leighton and Peter W. Shor, editors, *STOC*, pages 654–663. ACM, 1997.

[KSW05] Fabian Kuhn, Stefan Schmid, and Roger Wattenhofer. A Self-Repairing Peer-to-Peer System Resilient to Dynamic Adversarial Churn. In *4th International Workshop on Peer-To-Peer Systems (IPTPS), Cornell University, Ithaca, New York, USA, Springer LNCS 3640*, February 2005.

[KW08] Javed I. Khan and Adam Wierzbicki. Introduction: Guest editors' introduction: Foundation of peer-to-peer computing. *Comput. Commun.*, 31(2):187–189, February 2008.

[Lar07] Erik Larkin. Storm Worm's virulence may change tactics. http://www.networkworld.com/news/2007/080207-black-hat-storm-worms-virulence.html, Agust 2007. Last accessed on June 11, 2012.

[LMSW06] Thomas Locher, Patrick Moor, Stefan Schmid, and Roger Wattenhofer. Free Riding in BitTorrent is Cheap. In *5th Workshop on Hot Topics in Networks (HotNets), Irvine, California, USA*, November 2006.

[LMSW07] Thomas Locher, Remo Meier, Stefan Schmid, and Roger Wattenhofer. Push-to-Pull Peer-to-Peer Live Streaming. In *21st International Symposium on Distributed Computing (DISC), Lemesos, Cyprus*, September 2007.

[MM02] Petar Maymounkov and David Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 53–65, London, UK, UK, 2002. Springer-Verlag.

[MNR02] Dahlia Malkhi, Moni Naor, and David Ratajczak. Viceroy: a scalable and dynamic emulation of the butterfly. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 183–192, New York, NY, USA, 2002. ACM.

[MS07] Peter Mahlmann and Christian Schindelhauer. *Peer-to-Peer Networks.* Springer, 2007.

[PRR97] C. Greg Plaxton, Rajmohan Rajaraman, and Andréa W. Richa. Accessing Nearby Copies of Replicated Objects in a Distributed Environment. In *SPAA*, pages 311–320, 1997.

[PU88] David Peleg and Eli Upfal. A tradeoff between space and efficiency for routing tables. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, STOC '88, pages 43–52, New York, NY, USA, 1988. ACM.

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, November 2001.

[RFH$^+$01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. *SIGCOMM Comput. Commun. Rev.*, 31(4):161–172, August 2001.

[SENB07] Moritz Steiner, Taoufik En-Najjary, and Ernst W. Biersack. Exploiting KAD: possible uses and misuses. *SIGCOMM Comput. Commun. Rev.*, 37(5):65–70, October 2007.

[SG05] Ramesh Subramanian and Brian D. Goodman. *Peer to Peer Computing: The Evolution of a Disruptive Technology.* IGI Publishing, Hershey, PA, USA, 2005.

[SK85] Nicola Santoro and Ramez Khatib. Labelling and Implicit Routing in Networks. *Comput. J.*, 28(1):5–8, 1985.

[SMK$^+$01] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, August 2001.

[SW05] Ralf Steinmetz and Klaus Wehrle, editors. *Peer-to-Peer Systems and Applications*, volume 3485 of *Lecture Notes in Computer Science.* Springer, 2005.

[Wit81] L. D. Wittie. Communication Structures for Large Networks of Microcomputers. *IEEE Trans. Comput.*, 30(4):264–273, April 1981.

[ZHS+04] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, 2004.

Chapter 12

Advanced Blockchain

In this chapter we study various advanced blockchain concepts, which are popular in research.

12.1 Selfish Mining

Satoshi Nakamoto suggested that it is rational to be altruistic, e.g., by always attaching newly found block to the longest chain. But is it true?

Definition 12.1 (Selfish Mining)**.** *A selfish miner hopes to earn the reward of a larger share of blocks than its hardware would allow. The selfish miner achieves this by temporarily keeping newly found blocks secret.*

Algorithm 12.2 Selfish Mining

```
Idea: Mine secretly, without immediately publishing newly found blocks
Let $d_p$ be the depth of the public blockchain
Let $d_s$ be the depth of the secretly mined blockchain
if a new block $b_p$ is published, i.e., $d_p$ has increased by 1 then
  if $d_p > d_s$ then
    Start mining on that newly published block $b_p$
  else if $d_p = d_s$ then
    Publish secretly mined block $b_s$
    Mine on $b_s$ and publish newly found block immediately
  else if $d_p = d_s - 1$ then
    Publish all secretly mined blocks
  end if
end if
```

Theorem 12.3 (Selfish Mining)**.** *It may be rational to mine selfishly, depending on two parameters α and γ, where α is the ratio of the mining power of the selfish miner, and γ is the share of the altruistic mining power the selfish miner can reach in the network if the selfish miner publishes a block right after seeing a newly published block. Precisely, the selfish miner share is*

$$\frac{\alpha(1-\alpha)^2(4\alpha+\gamma(1-2\alpha))-\alpha^3}{1-\alpha(1+(2-\alpha)\alpha)}.$$

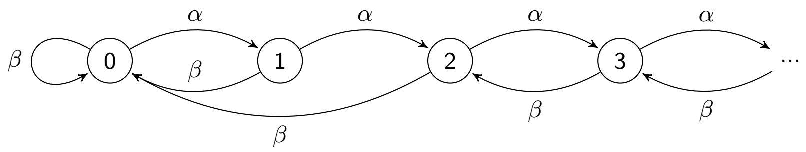


Figure 12.4: Each state of the Markov chain represents how many blocks the selfish miner is ahead, i.e., $d_s - d_p$. In each state, the selfish miner finds a block with probability α, and the honest miners find a block with probability $\beta = 1 - \alpha$. The interesting cases are the "irregular" β arrow from state 2 to state 0, and the β arrow from state 1 to state 0 as it will include three subcases.

Proof. We model the current state of the system with a Markov chain, see Figure 12.4.

We can solve the following Markov chain equations to figure out the probability of each state in the stationary distribution:

$$\begin{aligned} p_1 &= \alpha p_0 \\ \beta p_{i+1} &= \alpha p_i, \text{ for all } i > 1 \\ \text{and } 1 &= \sum_i p_i. \end{aligned}$$

Using $\rho = \alpha/\beta$, we express all terms of above sum with p_1:

$$1 = \frac{p_1}{\alpha} + p_1 \sum_{i \geq 0} \rho^i = \frac{p_1}{\alpha} + \frac{p_1}{1-\rho}, \text{ hence } p_1 = \frac{2\alpha^2 - \alpha}{\alpha^2 + \alpha - 1}.$$

Each state has an outgoing arrow with probability β. If this arrow is taken, one or two blocks (depending on the state) are attached that will eventually end up in the main chain of the blockchain. In state 0 (if arrow β is taken), the honest miners attach a block. In all states i with $i > 2$, the selfish miner eventually attaches a block. In state 2, the selfish miner directly attaches 2 blocks because of Line 11 in Algorithm 12.2.

State 1 in Line 8 is interesting. The selfish miner secretly was 1 block ahead, but now (after taking the β arrow) the honest miners are attaching a competing block. We have a race who attaches the next block, and where. There are three possibilities:

- Either the selfish miner manages to attach another block to its own block, giving 2 blocks to the selfish miner. This happens with probability α.
- Or the honest miners attach a block (with probability β) to their previous honest block (with probability $1 - \gamma$). This gives 2 blocks to the honest miners, with total probability $\beta(1 - \gamma)$.
- Or the honest miners attach a block to the selfish block, giving 1 block to each side, with probability $\beta\gamma$.

The blockchain process is just a biased random walk through these states. Since blocks are attached whenever we have an outgoing β arrow, the total number of blocks being attached per state is simply $1+p_1+p_2$ (all states attach a single block, except states 1 and 2 which attach 2 blocks each).

As argued above, of these blocks, $1-p_0+p_2+\alpha p_1-\beta(1-\gamma)p_1$ are blocks by the selfish miner, i.e., the ratio of selfish blocks in the blockchain is

$$\frac{1-p_0+p_2+\alpha p_1-\beta(1-\gamma)p_1}{1+p_1+p_2}.$$

□

Remarks:

- If the miner is honest (altruistic), then a miner with computational share α should expect to find an α fraction of the blocks. For some values of α and γ the ratio of Theorem 12.3 is higher than α.
- In particular, if $\gamma=0$ (the selfish miner only wins a race in Line 8 if it manages to mine 2 blocks in a row), the break even of selfish mining happens at $\alpha=1/3$.
- If $\gamma=1/2$ (the selfish miner learns about honest blocks very quickly and manages to convince half of the honest miners to mine on the selfish block instead of the slightly earlier published honest block), already $\alpha=1/4$ is enough to have a higher share in expectation.
- And if $\gamma=1$ (the selfish miner controls the network, and can hide any honest block until the selfish block is published) any $\alpha>0$ justifies selfish mining.

12.2 DAG-Blockchain

Traditional Bitcoin-like blockchains require mining blocks sequentially. Sometimes effort is wasted if two blocks happen to be mined at roughly the same time, as one of these two blocks is going to become obsolete. DAG-blockchains (where DAG stands for directed acyclic graph) try to prevent such wasted blocks. They allow for faster block production, as forks are less of a problem.

Definition 12.5 (DAG-blockchain). *In a DAG-blockchain the genesis block does not reference other blocks. Every other block has at least one (and possibly multiple references) to previous blocks. All references should be relevant in the sense that a reference does not already include another reference recursively. E.g., if block b refers to block b', and block b' refers to block b'', then block b does not (need to) include an explicit reference to block b'', since block b already includes an implicit recursive reference to b'' through b.*

Definition 12.6 (DAG-Relations). *Block p is a dag-parent of block b if block b references (includes a hash) to p. Likewise b is a dag-child of p. Block a is a dag-ancestor of block b, if a is b's dag-parent, dag-grandparent (dag-parent of dag-parent), dag-grandgrandparent, and so on. Likewise b is a's dag-descendant.*

Theorem 12.7. *There are no cycles in a DAG-blockchain.*

Proof. A block b includes its dag-parents' hashes. These dag-parents themselves include the hashes of their dag-parents, etc. To get a cycle of references, some of b's dag-ancestors must include b's hash, which is cryptographically infeasible. □

Definition 12.8 (Tree-Relations)**.** *We are going to implicitly mark some of the references in the DAG of blocks, such that these marked references form a tree, directed towards the genesis block. For every non-genesis block one edge to one of its dag-parents is marked. We use the prefix "tree" to denote these special relations. The marked edge is between tree-parent and tree-child. The tree also defines tree-ancestors and tree-descendants.*

Remarks:

- In other words, every tree-something is also a dag-something, but not necessarily vice versa.
- Blocks do not specify who is their tree-parent, or the order of their dag-parents. Instead, tree-parents are implicitly defined as follows.

Definition 12.9 (DAG Weight)**.** *The weight of a dag-ancestor block a with respect to a block b is defined as the number of tree-descendants of a in the set of dag-ancestors of b. If two blocks a and a' have the same weight, we use the hashes of a and a' to break ties.*

Definition 12.10 (Parent Order)**.** *Let x and y be any pair of dag-parents of b, and z be the lowest common tree-ancestor of x and y. x' and y' are the tree-children of z that are tree-ancestors of x and y respectively. If x' has a higher weight than y', then block b orders dag-parent x before y.*

Definition 12.11 (Tree-Parent)**.** *The tree-parent of b is the first dag-parent in b's parent order.*

Remarks:

- Now we can totally order all the blocks in the DAG-Blockchain.

Algorithm 12.12 DAG-Blockchain Ordering

```
We totally order all dag-ancestors of block $b$ as $<_b$ as follows:
Initialize $<_b$ as empty
for all dag-parents $p$ of $b$, in their parent order do
    Compute $<_p$ (recursively)
    Remove from $<_p$ any blocks already included in $<_b$
    Append $<_p$ at the end of $<_b$
end for
Append block $b$ at the end of $<_b$
```

Theorem 12.13. *Let p be the tree-parent of b. The order of blocks $<_b$ computed by Algorithm 12.12 extends the order $<_p$ by appending some blocks.*

Proof. Block p is the first dag-parent of b, so in the first iteration of the loop, we have $<_b = <_p$. Further modifications of $<_b$ consist only of appending more blocks to $<_b$, ending with block b itself. □

Remarks:

- Note that b is appended to the order only after ordering all its dag-ancestors. The genesis block is the only block where the recursion will stop, so the genesis block is always first in the total order.
- By Theorem 12.13 tree-children extend the order of their tree-parent, so appending blocks to the DAG preserves the previous order and new blocks are appended at the end.

Definition 12.14 (Transaction Order)**.** *Transactions in each block are ordered by the miner of the block. Since blocks themselves are ordered, all transactions are ordered. If two transactions contradict each other (e.g. they try to spend the same money twice), the first transaction in the total order is considered executed, while the second transaction is simply ignored (or possibly punished).*

Remarks:

- Ethereum allows blocks to not only have a parent, but also up to two "uncles" (childless blocks). In contrast to above description, blocks must specify the main parent.
- In Ethereum, new blocks are mined approximately every 15 seconds (as opposed to 10 minutes in Bitcoin). New blocks being generated in such rapid succession leads to a lot of childless blocks. Uncles have been introduced to not "waste" those blocks.
- In Ethereum, the original uncle-miners get 7/8 of the block reward. The miner who references these uncle blocks also gets a small reward. This reward depends on the height-difference of the uncle and the included parent. Also, to be included, the uncle and the current block should have a common ancestor not too far in the past.

12.3 Ethereum

Definition 12.15 (Ethereum)**.** *Ethereum is a distributed state machine. Unlike Bitcoin, Ethereum promises to run arbitrary computer programs in a blockchain.*

Remarks:

- Like the Bitcoin network, Ethereum consists of nodes that are connected by a random virtual network. These nodes can join or leave the network arbitrarily. There is no central coordinator.
- Like in Bitcoin, users broadcast cryptographically signed transactions in the network. Nodes collate these transactions and decide on the ordering of transactions by putting them in a block on the Ethereum blockchain.

Definition 12.16 (Smart Contract)**.** *Smart contracts are programs deployed on the Ethereum blockchain that have associated storage and can execute arbitrarily complex logic.*

Remarks:

- Smart Contracts are written in higher level programming languages like Solidity, Vyper, etc. and are compiled down to EVM (Ethereum Virtual Machine) bytecode, which is a Turing complete low level programming language.

- Smart contracts cannot be changed after deployment. But most smart contracts contain mutable storage, and this storage can be used to adapt the behavior of the smart contract. With this, many smart contracts can update to a new version.

Definition 12.17 (Account)**.** *Ethereum knows two kinds of accounts. Externally Owned Accounts (EOAs) are controlled by individuals, with a secret key. Contract Accounts (CAs) are for smart contracts. CAs are not controlled by a user.*

Definition 12.18 (Ethereum Transaction)**.** *An Ethereum transaction is sent by a user who controls an EOA to the Ethereum network. A transaction contains:*

- *Nonce: This "number only used once" is simply a counter that counts how many transactions the account of the sender of the transaction has already sent.*

- *160-bit address of the recipient.*

- *The transaction is signed by the user controlling the EOA.*

- *Value: The amount of Wei (the native currency of Ethereum) to transfer from the sender to the recipient.*

- *Data: Optional data field, which can be accessed by smart contracts.*

- *StartGas: A value representing the maximum amount of computation this transaction is allowed to use.*

- *GasPrice: How many Wei per unit of Gas the sender is paying. Miners will probably select transactions with a higher GasPrice, so a high GasPrice will make sure that the transaction is executed more quickly.*

Remarks:

- There are three types of transactions.

Definition 12.19 (Simple Transaction)**.** *A simple transaction in Ethereum transfers some of the native currency, called Wei, from one EOA to another. Higher units of curency are called Szabo, Finney, and Ether, with* 10^{18} *Wei =* 10^6 *Szabo =* 10^3 *Finney = 1 Ether. The data field in a simple transaction is empty.*

Definition 12.20 (Smart Contract Creation Transaction)**.** *A transaction whose recipient address field is set to* 0 *and whose data field is set to compiled EVM code is used to deploy that code as a smart contract on the Ethereum blockchain. The contract is considered deployed after it has been mined in a block and is included in the blockchain at a sufficient depth.*

Definition 12.21 (Smart Contract Execution Transaction)**.** *A transaction that has a smart contract address in its recipient field and code to execute a specific function of that contract in its data field.*

Remarks:

- Smart Contracts can execute computations, store data, send Ether to other accounts or smart contracts, and invoke other smart contracts.

- Smart contracts can be programmed to self destruct. This is the only way to remove them again from the Ethereum blockchain.

- Each contract stores data in 3 separate entities: storage, memory, and stack. Of these, only the storage area is persistent between transactions. Storage is a key-value store of 256 bit words to 256 bit words. The storage data is persisted in the Ethereum blockchain, like the hard disk of a traditional computer. Memory and stack are for intermediate storage required while running a specific function, similar to RAM and registers of a traditional computer. The read/write gas costs of persistent storage is significantly higher than those of memory and stack.

Definition 12.22 (Gas)**.** *Gas is the unit of an atomic computation, like swapping two variables. Complex operations use more than 1 Gas, e.g., ADDing two numbers costs 3 Gas.*

Remarks:

- As Ethereum contracts are programs (with loops, function calls, and recursions), end users need to pay more gas for more computations. In particular, smart contracts might call another smart contract as a subroutine, and StartGas must include enough gas to pay for all these function calls invoked by the transaction.

- The product of StartGas and GasPrice is the maximum cost of the entire transaction.

- Transactions are an all or nothing affair. If the entire transaction could not be finished within the StartGas limit, an Out-of-Gas exception is raised. The state of the blockchain is reverted back to its values before the transaction. The amount of gas consumed is not returned back to the sender.

Definition 12.23 (Block)**.** *In Ethereum, like in Bitcoin, a block is a collection of transactions that is considered a part of the canonical history of transactions. Among other things, a block contains: pointers to parent and up to two uncles, the hash of the root node of a trie structure populated with each transaction of the block, the hash of the root node of the state trie (after transactions have been executed)*

12.4 Payment Hubs

How to enable many parties to send payments to each other efficiently?

Definition 12.24 (Payment Hub). *Multiple parties can send payments to each other by means of a payment hub.*

Remarks:

- While we could always call the smart contract to transfer money between users that joined the hub, every smart contract call costs as it involves the blockchain. Rather, we want a frugal system with just few blockchain transactions.

Definition 12.25 (Smart Contract Hub). *A smart contract hub is a payment hub that is realized by a smart contract on a blockchain and an off-chain server. The smart contract and the server together enable off-chain payments between users that joined the hub.*

Algorithm 12.26 Smart Contract Hub

```
Users join the hub by depositing some native currency of the blockchain into the smart contract
Funds of all participants are maintained together as a fungible pool in the smart contract
Time is divided into epochs: in each epoch users can send each other payment transactions through the server
The server does the bookkeeping of who has paid how much to whom during the epoch
At the end of the epoch, the server aggregates all balances into a commitment, which is sent to the smart contract
Also at the end of the epoch, the server sends a proof to each user, informing about the current account balance
Each user can verify that its balance is correct; if not the user can call the smart contract with its proof to get its money back
```

Remarks:

- The smart contract lives forever, but the server can disappear anytime. If it does, nodes can show their recent balance proofs to the smart contract and withdraw their balances.

- The server can be scaled to in terms of latency and number of users. The smart contract does not need to scale as it only needs to just accept one commitment per epoch.

- In case the server disappears, the smart contract will be flooded with withdrawal requests, and could be subject to delays based on the delays of the underlying blockchain.

12.5 Proof-of-Stake

Almost all of the energy consumption of permissionless (everybody can participate) blockchains is wasted because of proof-of-work. Proof-of-stake avoids these wasteful computations, without going all the way to permissioned (the participating nodes are known a priori) systems such as Paxos or PBFT.

Definition 12.27 (Proof-of-stake)**.** *Proof-of-work awards block rewards to the lucky miner that solved a cryptopuzzle. In contrast, proof-of-stake awards block rewards proportionally to the economic stake in the system.*

Remarks:

- Literally, "the rich get richer".
- Ethereum is expected to move to proof-of-stake eventually.
- There are multiple flavors of proof-of-stake algorithms.

Definition 12.28 (Chain based proof-of-stake)**.** *Accounts hold lottery tickets according to their stake. The lottery is pseudo-random, in the sense that hash functions computed on the state of the blockchain will select which account is winning. The winning account can extend the longest chain by a block, and earn the block reward.*

Remarks:

- It gets tricky if the actual winner of the lottery does not produce a block in time, or some nodes do not see this block in time. This is why some suggested proof-of-stake systems add a voting phase.

Definition 12.29 (BFT based proof-of-stake)**.** *The lottery winner only gets to propose a block to be added to the blockchain. A committee then votes (yes, byzantine fault tolerance) whether to accept that block into the blockchain. If no agreement is reached, this process is repeated.*

Remarks:

- Proof-of-stake can be attacked in various ways. Let us discuss the two most prominent attacks.
- Most importantly, there is the "nothing at stake" attack: In blockchains, forks occur naturally. In proof-of-work, a fork is resolved because every miner has to choose which blockchain fork to extend, as it does not pay off to mine on a hopeless fork. Eventually, some chain will end up with more miners, and that chain is considered to be the real blockchain, whereas other (childless) blocks are just not being extended. In a proof-of-stake system, a user can trivially extend all prongs of a fork. As generating a block costs nothing, the miner has no incentive to not extend all the prongs of the fork. This results in a situation with more and more forks, and no canonical order of transactions. If there is a double-spend attack, there is no way to tell which blockchain is valid, as all blockchains are the same length (all

miners are extending all forks). It can be argued that honest miners, who want to preserve the value of the network, will extend the first prong of the fork that they see. But that leaves room for a dishonest miner to double spend by moving their mining opportunity to the appropriate fork at the appropriate time.

- Long range attack: As there are no physical resources being used to produce blocks in a proof-of-stake system, nothing prevents a bad player from creating an alternate blockchain starting at the genesis block, and make it longer than the canonical blockchain. New nodes may have difficulties to determine which blockchain is the real established blockchain. In proof-of-work, long range attacks takes an enormous amount of computing power. In proof-of-stake systems, a new node has to check with trusted sources to know what the canonical blockchain is.

Chapter Notes

Selfish mining has already been discussed shortly after the introduction of Bitcoin [RHo10]. A few years later, Eyal and Sirer formally analyzed selfish mining [ES14]. If the selfish miner is two or more blocks ahead, this original research suggested to always answer a newly published block by releasing the oldest unpublished block, so have two blocks at the same level. The idea was that honest miners will then split their mining power between these two blocks. However, what matters is how long it takes the honest miners to find the next block to extend the public blockchain. This time does not change whether the honest miners split their efforts or not. Hence the case $d_p < d_s - 1$ is not needed in Algorithm 12.2.

Similarly, Courtois and Bahack [CB14] study subversive mining strategies. Nayak et al. [NKMS15] combine selfish mining and eclipse attacks. Algorithm 12.2 is not optimal for all parameters, e.g., sometimes it may be beneficial to risk even a two-block advantage. Sapirshtein et al. [SSZ15] describe and analyze the optimal algorithm.

Vitalik Buterin introduced Ethereum in the 2013 whitepaper [But13]. In 2014, Ethereum Foundation was founded to create Ethereum's first implementation. An online crowd-sale was conducted to raise around 31,000 BTC (around USD 18 million at the time) for this. In this sense, Ethereum was the first ICO (Initial Coin Offering). Ethereum has also attempted to write a formal specification of its protocol in their yellow paper [Gav18]. This is in contrast to Bitcoin, which doesn't have a formal specification.

Bitcoin's blockchain forms as a chain, i.e., each block (except the genesis block) has a parent block. The longest chain with the highest difficulty is considered the main chain. GHOST [SZ15] is an alternative to the longest chain rule for establishing consensus in PoW based blockchains and aims to alleviate adverse impacts of stale blocks. Ethereum's blockchain structure is a variant of GHOST. Other systems based on DAGs have been proposed in [SLZ16], [SZ18], [LLX$^+$18], and [LSZ15].

Bidirectional payment channels were introduced in [DW15] and [PD15]. Multiple such channels can be linked together to form an overlay network on top of the Bitcoin network to enable off-chain payments. Payments between nodes that do not have a channel between them is done by modifying the commitment transaction to add a third output that is unlocked with a secret that is shared over multiple hops by multiple channels. Lightning has a few implementations with active development: c-lightning, lnd, and eclair. These support lightning based wallets and node software, and depend on a co-located Bitcoin full node.

Khalil and Gervais [KG18] introduced the notion of a payment hub that is a combination of a smart contract and an online server. Plasma [JP17] is another family of systems that uses a smart contract on the Ethereum blockchain and one or more off-chain operators to enable off-chain transactions.

Proof of Stake was first introduced in PPCoin [KN12]. The most well known Proof of Stake system is the one being implemented for Ethereum, which involves a transition phase from PoW to PoS [BG17], and finally, on to a more formally constructed PoS [VZ18]. More details are available in this article [Cho18] by Jon Choi.

Bibliography

[BG17] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget. *CoRR*, abs/1710.09437, 2017.

[But13] Vitalik Buterin. A Next-Generation Smart Contract and Decentralized Application Platform, 2013. Available from: `https://github.com/ethereum/wiki/wiki/White-Paper`.

[CB14] Nicolas T. Courtois and Lear Bahack. On subversive miner strategies and block withholding attack in bitcoin digital currency. *CoRR*, abs/1402.1718, 2014.

[Cho18] Jon Choi. Ethereum casper 101. 2018. Available from: `https://medium.com/@jonchoi/ethereum-casper-101-7a851a4f1eb0`.

[DW15] Christian Decker and Roger Wattenhofer. A Fast and Scalable Payment Network with Bitcoin Duplex Micropayment Channels. In *Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS)*, 2015.

[ES14] Ittay Eyal and Emin Gün Sirer. Majority is not enough: Bitcoin mining is vulnerable. In *Financial Cryptography and Data Security*, pages 436–454. Springer, 2014.

[Gav18] Gavin Wood. Ethereum: A Secure Decentralised Generalised Transaction Ledger, Byzantium Version, 2018. Available from: `https://ethereum.github.io/yellowpaper/paper.pdf`.

[JP17] Vitalik Buterin Joseph Poon. Plasma: Scalable autonomous smart contracts, 2017.

[KG18] Rami Khalil and Arthur Gervais. Nocust - a non-custodial 2nd-layer financial intermediary. Cryptology ePrint Archive, Report 2018/642, 2018. `https://eprint.iacr.org/2018/642`.

[KN12] Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. *self-published paper, August*, 19, 2012.

[LLX$^+$18] Chenxing Li, Peilun Li, Wei Xu, Fan Long, and Andrew Chi-Chih Yao. Scaling nakamoto consensus to thousands of transactions per second. *CoRR*, abs/1805.03870, 2018.

[LSZ15] Yoad Lewenberg, Yonatan Sompolinsky, and Aviv Zohar. Inclusive block chain protocols. In *Financial Cryptography and Data Security*, pages 528–547. Springer, 2015.

[NKMS15] Kartik Nayak, Srijan Kumar, Andrew Miller, and Elaine Shi. Stubborn mining: Generalizing selfish mining and combining with an eclipse attack. Technical report, IACR Cryptology ePrint Archive 2015, 2015.

[PD15] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network. 2015.

[RHo10] RHorning. Mining cartel attack, 2010.

[SLZ16] Yonatan Sompolinsky, Yoad Lewenberg, and Aviv Zohar. Spectre: A fast and scalable cryptocurrency protocol. Cryptology ePrint Archive, Report 2016/1159, 2016. `https://eprint.iacr.org/2016/1159`.

[SSZ15] Ayelet Sapirshtein, Yonatan Sompolinsky, and Aviv Zohar. Optimal selfish mining strategies in bitcoin. *arXiv preprint arXiv:1507.06183*, 2015.

[SZ15] Yonatan Sompolinsky and Aviv Zohar. Secure high-rate transaction processing in bitcoin. In *Financial Cryptography and Data Security*, pages 507–527. Springer, 2015.

[SZ18] Yonatan Sompolinsky and Aviv Zohar. Phantom: A scalable blockdag protocol. Cryptology ePrint Archive, Report 2018/104, 2018. `https://eprint.iacr.org/2018/104`.

[VZ18] Aditya Asgaonkar Georgios Piliouras Vlad Zamfir, Nate Rush. Introducing the “minimal cbc casper” family of consensus protocols. 2018. Available from: `https://github.com/cbc-casper/cbc-casper-paper/blob/master/cbc-casper-paper-draft.pdf`.

Chapter 13

Authenticated Agreement

In Section 5.4 we have already had a glimpse into the power of cryptography. In this Chapter we want to build a *practical* byzantine fault-tolerant system using cryptography. With cryptography, byzantine lies may be detected easily.

13.1 Agreement with Authentication

Definition 13.1 (Signature)**.** *Every node can digitally* ***sign*** *its messages in a way that no other node can forge, thus nodes can reliably determine which node a signed message originated from. We denote a message* `msg`(x) *signed by node* u *with* `msg`$(x)_u$.

Algorithm 13.2 Byzantine Agreement with Authentication

```
    Code for primary p:
if input is 1 then
    broadcast value$(1)_p$
    decide 1 and terminate
else
    decide 0 and terminate
end if
    Code for all other nodes v:
for all rounds $i \in \{1, \dots, f+1\}$ do
    $S$ is the set of accepted messages value$(1)_u$.
    if $|S| \geq i$ and value$(1)_p \in S$ then
        broadcast $S \cup \{$value$(1)_v\}$
        decide 1 and terminate
    end if
end for
decide 0 and terminate
```

Remarks:

- Algorithm 13.2 solves byzantine agreement on binary inputs relying on signatures. We assume there is a designated "primary" node p that all other nodes know. The goal is to decide on p's value.

Theorem 13.3. *Algorithm 13.2 can tolerate $f < n$ byzantine failures while terminating in $f + 1$ rounds.*

Proof. Assuming that the primary p is not byzantine and its input is 1, then p broadcasts $\texttt{value}(1)_p$ in the first round, which will trigger all correct nodes to decide on 1. If p's input is 0, there is no signed message $\texttt{value}(1)_p$, and no node can decide on 1.

If primary p is byzantine, we need all correct nodes to decide on the same value for the algorithm to be correct.

Assume $i < f + 1$ is the minimal round in which any correct node u decides on 1. In this case, u has a set S of at least i messages from other nodes for value 1 in round i, including one of p. Therefore, in round $i + 1 \leq f + 1$, all other correct nodes will receive S and u's message for value 1 and thus decide on 1 too.

Now assume that $i = f + 1$ is the minimal round in which a correct node u decides for 1. Thus u must have received $f + 1$ messages for value 1, one of which must be from a correct node since there are only f byzantine nodes. In this case some other correct node v must have decided on 1 in some round $j < i$, which contradicts i's minimality; hence this case cannot happen.

Finally, if no correct node decides on 1 by the end of round $f + 1$, then all correct nodes will decide on 0. □

Remarks:

- If the primary is a correct node, Algorithm 13.2 only needs two rounds! Otherwise, the algorithm terminates in at most $f + 1$ rounds, which is optimal as described in Theorem 4.20.

- By using signatures, Algorithm 13.2 manages to solve consensus for any number of failures! Does this contradict Theorem 4.12? Recall that in the proof of Theorem 4.12 we assumed that a byzantine node can distribute contradictory information about its own input. If messages are signed, correct nodes can detect such behavior. Specifically, if a node u signs two contradicting messages, then observing these two messages proves to all nodes that node u is byzantine.

- Does Algorithm 13.2 satisfy any of the validity conditions introduced in Section 4.1? No! A byzantine primary can dictate the decision value.

- Can we modify the algorithm such that the correct-input validity condition is satisfied? Yes! We can run the algorithm in parallel for $2f+1$ primary nodes. Either 0 or 1 will occur at least $f + 1$ times, which means that one correct process had to have this value in the first place. In this case, we can only handle $f < \frac{n}{2}$ byzantine nodes.

- Can we make it work with arbitrary inputs?

- Relying on synchrony limits the practicality of the protocol. What if messages can be lost or the system is asynchronous?

13.2 Practical Byzantine Fault Tolerance

Practical Byzantine Fault Tolerance (PBFT) is one of the first and perhaps the most instructive protocol for achieving state replication among nodes as in Definition 2.8 with byzantine nodes in an asynchronous network. We present a simplified version of PBFT without any optimizations.

Definition 13.4 (System Model)**.** *We consider a system with $n = 3f + 1$ nodes, and additionally an unbounded number of clients. There are at most f byzantine nodes, and clients can be byzantine as well. The network is asynchronous, and messages have variable delay and can get lost. Clients send requests that correct nodes have to order to achieve state replication.*

Remarks:

- At any given time, every node will consider one designated node to be the *primary* and the other nodes to be *backups*.
- The timespan for which a node p is seen as the primary from the perspective of another node is called a view.

Definition 13.5 (View)**.** *A* ***view*** *v is a non-negative integer representing the node's local perception of the system. We say that* ***node*** *u* ***is in view*** *v as long as node u considers node $p = v \bmod n$ to be the primary.*

Remarks:

- All nodes start out in view 0. Nodes can potentially be in different views (i.e. have different local values for v) at any given time.
- If backups detect faulty behavior in the primary, they switch to the next primary with a so-called *view change* (see Section 13.4).
- In the asynchronous model, requests can arrive at the nodes in different orders. While a primary remains in charge (sufficiently many nodes share the view v), it thus adopts the function of a serializer (cf. Algorithm 2.9).

Definition 13.6 (Sequence Number)**.** *During a view, a node relies on the primary to assign consecutive* ***sequence numbers*** *(integers) that function as indices in the global order (cf. Definition 2.8) for the requests that clients send.*

Remarks:

- During a view change, we ensure that no two correct nodes execute requests in different orders. On the one hand, we need to exchange information on the current state to guarantee that a correct new primary knows the latest sequence number that has been accepted by sufficiently many backups. On the other hand, exchanging information will enable backups to determine if the new primary acts in a byzantine fashion, e.g. reassigning the latest sequence number to a different request.

- We use signatures to guarantee that every node can determine which node/client has generated any given message.

Definition 13.7 (Accepted Messages)**.** *A correct node that is in view v will only* ***accept*** *messages that it can authenticate, that follow the specification of the protocol, and that also belong to view v.*

Remarks:

- The protocol will guarantee that once a correct node has executed a request r with sequence number s, then no correct node will execute any request $r' \neq r$ with sequence number s, not unlike Lemma 2.14.
- Correct primaries choose sequence numbers in order, without gap, i.e. if a correct primary proposed s as the sequence number for the last request, then it will use $s+1$ for the next request that it proposes.
- Before a node can safely execute a request r with a sequence number s, it will wait until it knows that the decision to execute r with s has been reached and is widely known.
- Informally, nodes will collect confirmation messages by sets of at least $2f+1$ nodes to guarantee that that information is sufficiently widely distributed.

Lemma 13.8 ($2f+1$ Quorum Intersection)**.** *Let S_1 with $|S_1| \geq 2f+1$ and S_2 with $|S_2| \geq 2f+1$ each be sets of nodes. Then there exists a correct node in $S_1 \cap S_2$.*

Proof. Let S_1, S_2 each be sets of at least $2f+1$ nodes. There are $3f+1$ nodes in total, thus due to the pigeonhole principle the intersection $S_1 \cap S_2$ contains at least $f+1$ nodes. Since there are at most f faulty nodes, $S_1 \cap S_2$ contains at least 1 correct node. □

13.3 PBFT: Agreement Protocol

First we describe how PBFT achieves agreement on a unique order of requests within a single view. Figure 13.9 shows how the nodes come to an agreement on a sequence number for a client request. Informally, the protocol has these five steps:

1. The nodes receive a request and relay it to the primary.
2. The primary sends a `pre-prepare`-message to all backups, informing them that it wants to execute that request with the sequence number specified in the message.
3. Backups send `prepare`-messages to all nodes, informing them that they agree with that suggestion.
4. All nodes send `commit`-messages to all nodes, informing everyone that they have committed to execute the request with that sequence number.
5. They execute the request and inform the client.

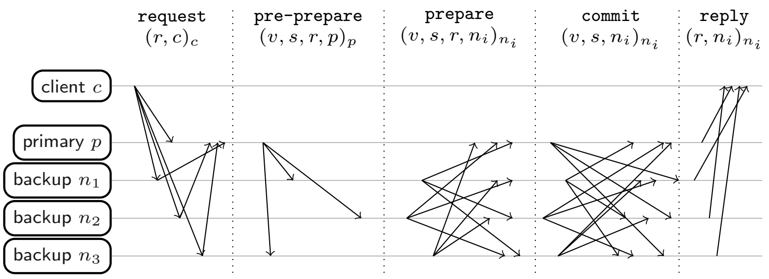


Figure 13.9: The agreement protocol used in PBFT for processing a client request issued by client c, exemplified for a system with $n = 4$ nodes. The primary in view v is $p = n_0 = v \bmod n$.

Remarks:

- To make sure byzantine nodes cannot force the execution of a request, every node waits for a certain number of `prepare`- and `commit`-messages with the correct content before executing the request.
- Definitions 13.10, 13.12, 13.14, and 13.16 specify the agreement protocol formally. Backups run the pre-prepare and the prepare phase concurrently.

Definition 13.10 (Pre-Prepare Phase)**.** *In the* ***pre-prepare phase*** *of the agreement protocol, the nodes execute Algorithm 13.11.*

Algorithm 13.11 PBFT Agreement Protocol: Pre-Prepare Phase

```
Code for primary p in view v:
accept request$(r, c)_c$ that originated from client $c$
pick next sequence number $s$
send pre-prepare$(v, s, r, p)_p$ to all backups
Code for backup b:
accept request$(r, c)_c$ from client $c$
relay request$(r, c)_c$ to primary $p$
```

Definition 13.12 (Prepare Phase)**.** *In the* ***prepare phase*** *of the agreement protocol, every backup b executes Algorithm 13.13. Once it has sent the* `prepare`*-message, we say that b has* ***pre-prepared*** *r for* (v, s)*.*

Remarks:

- What if a byzantine primary does not send a `pre-prepare` message for a request?

Algorithm 13.13 PBFT Agreement Protocol: Prepare Phase

```
Code for backup $b$ in view $v$:
accept pre-prepare$(v, s, r, p)_p$
if $b$ has not yet accepted a pre-prepare-message for $(v, s, r')$ with $r' \neq r$ then
    send prepare$(v, s, r, b)_b$ to all nodes
end if
```

Definition 13.14 (Prepared-Certificate)**.** *A node n_i that has pre-prepared a request executes Algorithm 13.15. It waits until it has collected $2f$* prepare-*messages (including n_i's own, if it is a backup) in Line 1. Together with the* pre-prepare-*message for (v, s, r), they form a **prepared-certificate**.*

Algorithm 13.15 PBFT Agreement Protocol: Commit Phase

```
Code for node $n_i$ that has pre-prepared $r$ for $(v, s)$:
1: wait until $2f$ prepare-messages matching $(v, s, r)$ have been accepted
2: create prepared-certificate for $(v, s, r)$
3: send commit$(v, s, n_i)_{n_i}$ to all nodes
```

Definition 13.16 (Committed-Certificate)**.** *A node n_i that has created a prepared-certificate for a request executes Algorithm 13.17. It waits until it has collected $2f + 1$* commit-*messages (including n_i's own) in Line 1. They form a **committed-certificate** and allow to safely execute the request once all requests with lower sequence numbers have been executed.*

Algorithm 13.17 PBFT Agreement Protocol: Execute Phase

```
Code for node $n_i$ that has created a prepared-certificate for $(v, s, r)$:
1: wait until $2f + 1$ commit-messages matching $(v, s)$ have been accepted
2: create committed-certificate for $(v, s, r)$
3: wait until all requests with lower sequence numbers have been executed
4: execute request $r$
5: send reply$(r, n_i)_{n_i}$ to client
```

Remarks:

- Note that the agreement protocol can run for multiple requests in parallel. Since we are in the variable delay model and messages can arrive out of order, we thus have to wait in Algorithm 13.17 Line 3 for all requests with lower sequence numbers to be executed.
- The client only considers the request to have been processed once it received $f + 1$ reply-messages sent by the nodes in Algorithm 13.17 Line 5. Since a correct node only sends a reply-message once it executed the request, with $f + 1$ reply-messages the client can be certain that the request was executed by a correct node.

- We will see in Section 13.4 that PBFT guarantees that once a single correct node executed the request, then all correct nodes will never execute a different request with the same sequence number. Thus, knowing that a single correct node executed a request is enough for the client.

- If the client does not receive at least $f+1$ `reply`-messages fast enough, it can start over by resending the request to initiate Algorithm 13.11 again. To prevent correct nodes that already executed the request from executing it a second time, clients can mark their requests with some kind of unique identifiers like a local timestamp. Correct nodes can then react to each request that is resent by a client as required by PBFT, and they can decide if they still need to execute a given request or have already done so before.

Lemma 13.18 (PBFT: Unique Sequence Numbers within View)**.** *If a node was able to create a prepared-certificate for (v, s, r), then no node can create a prepared-certificate for (v, s, r') with $r' \neq r$.*

Proof. Assume two (not necessarily distinct) nodes create prepared-certificates for (v, s, r) and (v, s, r'). Since a prepared-certificate contains $2f+1$ messages, by Lemma 13.8, some correct node must have sent a `pre-prepare`- or `prepare`-message both matching (v, s, r) and (v, s, r'). However, a correct primary only sends a single `pre-prepare`-message for each (v, s), see Algorithm 13.11 Lines 2 and 3. Furthermore, a correct backup only sends a single `prepare`-message for each (v, s), see Algorithm 13.13 Lines 2 and 3. Thus, $r' = r$. □

Remarks:

- Due to Lemma 13.18, once a node has a prepared-certificate for (v, s, r), no correct node will execute some $r' \neq r$ with sequence number s during view v because correct nodes wait for a prepared-certificate before executing a request (cf. Algorithm 13.15).

- However, that is not yet enough to make sure that no $r' \neq r$ will be executed by a correct node with sequence number s during some later view $v' > v$. How can we make sure that that does not happen?

13.4 PBFT: View Change Protocol

If the primary is faulty, the system has to perform a view change to move to the next primary so the system can make progress. To that end, nodes use a local faulty-timer (and only that!) to decide whether they consider the primary to be faulty.

Definition 13.19 (Faulty-Timer)**.** *When backup b accepts request r in Algorithm 13.11 Line 4, b starts a local **faulty-timer** (if the timer is not already running) that will only stop once b executes r.*

Remarks:

- If the faulty-timer expires, the backup considers the primary faulty and triggers a view change. When triggering a view change, a correct node will no longer participate in the protocol for the current view.
- We leave out the details regarding for what timespan to set the faulty-timer. This is a patience trade-off (more patience: slower turnover if the primary is byzantine; less patience: risk of prematurely firing view changes).
- During a view change, the protocol has to guarantee that requests that have already been executed by some correct nodes will not be executed with the different sequence numbers by other correct nodes.
- How can we guarantee that this happens?

Definition 13.20 (PBFT: View Change Protocol)**.** *In the view change protocol, a node whose faulty-timer has expired enters the* ***view change phase*** *by running Algorithm 13.22. During the* ***new view phase*** *(which all nodes continuously listen for), the primary of the next view runs Algorithm 13.24 while all other nodes run Algorithm 13.25.*

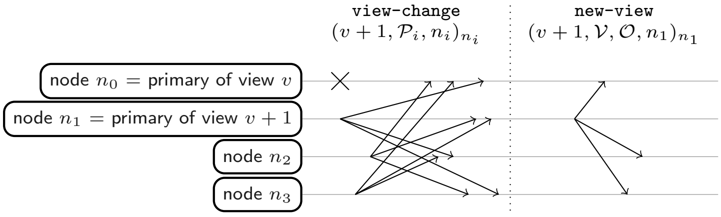

view-change
$(v+1, \mathcal{P}_i, n_i)_{n_i}$
new-view
$(v+1, \mathcal{V}, \mathcal{O}, n_1)_{n_1}$
node n_0 = primary of view v
node n_1 = primary of view $v+1$
node n_2
node n_3

Figure 13.21: The view change protocol used in PBFT. Node n_0 is the primary of current view v, node n_1 the primary of view $v+1$. Once backups consider n_0 to be faulty, they start the view change protocol (cf. Algorithms 13.22, 13.24, 13.25). The X signifies that n_0 is faulty.

Remarks:

- The idea behind the view change protocol is as follows: during the view change protocol, the new primary collects prepared-certificates from $2f+1$ nodes, so for every request that some correct node executed, the new primary will have at least one prepared-certificate.
- After gathering that information, the primary distributes it and tells all backups which requests need to be to executed with which sequence numbers.
- Backups can check whether the new primary makes the decisions required by the protocol, and if it does not, then the new primary must be byzantine and the backups can directly move to the next view change.

Algorithm 13.22 PBFT View Change Protocol: View Change Phase

```
  Code for node $n_i$ in view $v$ whose faulty-timer has expired:
stop accepting pre-prepare/prepare/commit-messages for $v$
let $\mathcal{P}_i$ be the set of all prepared-certificates that $n_i$ has collected since the system was started
send view-change$(v+1, \mathcal{P}_i, n_i)_{n_i}$ to all nodes
```

Definition 13.23 (New-View-Certificate). *$2f+1$ `view-change`-messages for the same view v form a **new-view-certificate**.*

Algorithm 13.24 PBFT View Change Protocol: New View Phase - Primary

```
  Code for new primary $p$ of view $v+1$:
1: accept $2f+1$ view-change-messages (including possibly $p$'s own) in a set $\mathcal{V}$ (this is the new-view-certificate)
2: let $\mathcal{O}$ be a set of pre-prepare$(v+1, s, r, p)_p$ for all pairs $(s, r)$ where at least one prepared-certificate for $(s, r)$ exists in $\mathcal{V}$
3: let $s^{\mathcal{V}}_{max}$ be the highest sequence number for which $\mathcal{O}$ contains a pre-prepare-message
4: add to $\mathcal{O}$ a message pre-prepare$(v+1, s', $null$, p)_p$ for every sequence number $s' < s^{\mathcal{V}}_{max}$ for which $\mathcal{O}$ does not contain a pre-prepare-message
5: send new-view$(v+1, \mathcal{V}, \mathcal{O}, p)_p$ to all nodes
6: start processing requests for view $v+1$ according to Algorithm 13.11 starting from sequence number $s^{\mathcal{V}}_{max}+1$
```

Remarks:

- It is possible that $\mathcal{V}$ contains a prepared-certificate for a sequence number s while it does not contain one for some sequence number $s' < s$. For each such sequence number s', we fill up $\mathcal{O}$ in Algorithm 13.24 Line 4 with `null`-requests, i.e. requests that backups understand to mean "do not do anything here".

Theorem 13.26 (PBFT:Unique Sequence Numbers Across Views). *Together, the PBFT agreement protocol and the PBFT view change protocol guarantee that if a correct node executes a request r in view v with sequence number s, then no correct node will execute any $r' \neq r$ with sequence number s in any view $v' \geq v$.*

Proof. If no view change takes place, then Lemma 13.18 proves the statement. Therefore, assume that a view change takes place, and consider view $v' > v$.

We will show that if some correct node executed a request r with sequence number s during v, then a correct primary will send a `pre-prepare`-message matching (v', s, r) in the $\mathcal{O}$-component of the `new-view`$(v', \mathcal{V}, \mathcal{O}, p)$-message. This guarantees that no correct node will be able to collect a prepared-certificate for s and a different $r' \neq r$.

Algorithm 13.25 PBFT View Change Protocol: New View Phase - Backup

Code for backup b of view $v+1$ if b's local view is $v' < v+1$:

```
 1: accept new-view$(v+1, \mathcal{V}, \mathcal{O}, p)_p$
 2: stop accepting pre-prepare-/prepare-/commit-messages for $v$
 3: set local view to $v+1$
 4: if $p$ is primary of $v+1$ then
 5:   if $\mathcal{O}$ was correctly constructed from $\mathcal{V}$ according to Algorithm 13.24
      Lines 2 and 4 then
 6:     respond to all pre-prepare-messages in $\mathcal{O}$ as in the agreement protocol,
        starting from Algorithm 13.13
 7:     start accepting messages for view $v+1$
 8:   else
 9:     trigger view change to $v+2$ using Algorithm 13.22
10:   end if
11: end if
```

Consider the new-view-certificate $\mathcal{V}$ (see Algorithm 13.24 Line 1). If any correct node executed request r with sequence number s, then due to Algorithm 13.17 Line 1, there is a set R_1 of at least $2f+1$ nodes that sent a `commit`-message matching (s, r), and thus the correct nodes in R_1 all collected a prepared-certificate in Algorithm 13.15 Line 1.

The new-view-certificate contains `view-change`-messages from a set R_2 of $2f+1$ nodes. Thus according to Lemma 13.8, there is at least one correct node $c_r \in R_1 \cap R_2$ that both collected a prepared-certificate matching (s, r) and whose `view-change`-message is contained in $\mathcal{V}$.

Therefore, if some correct node executed r with sequence number s, then $\mathcal{V}$ contains a prepared-certificate matching (s, r) from c_r. Thus, if some correct node executed r with sequence number s, then due to Algorithm 13.24 Line 2, a correct primary p sends a `new-view`$(v', \mathcal{V}, \mathcal{O}, p)$-message where $\mathcal{O}$ contains a `pre-prepare`(v', s, r, p)-message.

Correct backups will enter view v' only if the `new-view`-message for v' contains a valid new-view-certificate $\mathcal{V}$ and if $\mathcal{O}$ was constructed correctly from $\mathcal{V}$, see Algorithm 13.25 Line 5. They will then respond to the messages in $\mathcal{O}$ before they start accepting other `pre-prepare`-messages for v' due to the order of Algorithm 13.25 Lines 6 and 7. Therefore, for the sequence numbers that appear in $\mathcal{O}$, correct backups will only send `prepare`-messages responding to the `pre-prepare`-messages found in $\mathcal{O}$ due to Algorithm 13.13 Lines 2 and 3. This guarantees that in v', for every sequence number s that appears in $\mathcal{O}$, backups can only collect prepared-certificates for the triple (v', s, r) that appears in $\mathcal{O}$.

Together with the above, this proves that if some correct node executed request r with sequence number s in v, then no node will be able to collect a prepared-certificate for some $r' \neq r$ with sequence number s in any view $v' \geq v$, and thus no correct node will execute r' with sequence number s. □

Remarks:

- We have shown that PBFT protocol guarantees safety or nothing bad ever happens, i.e., the correct nodes never disagree on requests that were commited with the same sequence numbers. But, does PBFT also guarantee liveness? In other words, will a legitimate client request eventually be committed and replied?

- To prove liveness, we need message delays to be finite and bounded. With unbounded message delays in an asynchronous system and even one faulty process, it is impossible to solve consensus with guaranteed termination (Theorem 3.14).

- A faulty new primary could delay the system indefinitely by never sending a `new-view`-message. To prevent this, as soon as a node sends its `view-change`-message for view $v+1$, it starts its faulty-timer. and stops it once it accepts a for $v+1$. If the timer fires before receiving the `new-view`-message, the node triggers another view change.

- Since message delays are unknown, timers are doubling with every view. Eventually, the timeout is larger than the maximum message delay, and all correct messages are received before any timer expires.

- Since at most f consecutive primaries can be faulty, the system makes progress after at most $f+1$ view changes.

- We described a simplified version of PBFT; any practically relevant variant makes adjustments to what we presented. The references found in the chapter notes can be consulted for details that we did not include.

Chapter Notes

PBFT is perhaps the central protocol for asynchronous byzantine state replication. The seminal first publication about it, of which we presented a simplified version, can be found in [CL^{+}99]. The canonical work about most versions of PBFT is Miguel Castro's PhD dissertation [Cas01].

Notice that the sets $\mathcal{P}_b$ in Algorithm 13.22 grow with each view change as the system keeps running since they contain all prepared-certificates that nodes have collected so far. All variants of the protocol found in the literature introduce regular *checkpoints* where nodes agree that enough nodes executed all requests up to a certain sequence number so they can continuously garbage-collect prepared-certificates. We left this out for conciseness.

Remember that all messages are signed. Generating signatures is somewhat pricy, and variants of PBFT exist that use the cheaper, but less powerful Message Authentication Codes (MACs). These variants are more complicated because MACs only provide authentication between the two endpoints of a message and cannot prove to a third party who created a message. An extensive treatment of a variant that uses MACs can be found in [CL02].

Before PBFT, byzantine fault-tolerance was considered impractical, just something academics would be interested in. PBFT changed that as it showed that byzantine fault-tolerance can be practically feasible. As a result, numerous asynchronous byzantine state replication protocols were developed. Other well-known protocols are Q/U [AEMGG+05], HQ [CML+06], and Zyzzyva [KAD+07]. An overview over the relevant literature can be found in [AGK+15].

This chapter was written in collaboration with Georg Bachmeier.

Bibliography

[AEMGG+05] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable byzantine fault-tolerant services. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 59–74. ACM, 2005.

[AGK+15] Pierre-Louis Aublin, Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. *ACM Transactions on Computer Systems (TOCS)*, 32(4):12, 2015.

[Cas01] Miguel Castro. *Practical Byzantine Fault Tolerance*. Ph.d., MIT, January 2001. Also as Technical Report MIT-LCS-TR-817.

[CL+99] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[CL02] Miguel Castro and Barbara Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, 2002.

[CML+06] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 177–190. USENIX Association, 2006.

[KAD+07] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 45–58. ACM, 2007.

Index