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

**Notes:**

- Bring Laptop for Paxos code
- After the 10' intro, explain what nodes, message passing (not shared memory), single client and single server, acks, multiple servers, multiple clients, +1 vs. *2, state replication with master-slave replication (serializer), two-phase protocols, discussion thereof, what is issue with locks (what if nobody can get it) -¿ tickets. In the last 2' the two properties of tickets
- after the break the naive ticket protocol, trying to get problems (many are on the table, including malicious clients, missing acks, which command will be excecuted, and at some point also main problem, executing the wrong command (prepare this point well). Then (maybe 25' left) Paxos with projector, explain the basic protocol, then going into proof (prepare this point well). They want to discuss how to have more than 1 command... we do not really go into liveness problem, no time for that. All in all, the lecture is good, but not spectacular. They seem to know a few things already from DB, e.g. Quorum.

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.

**Notes:** Example: NAS at home, want to store/manipulate files. Assume that we can send a whole file in one message.

**Algorithm 2.3** Naïve Client-Server Algorithm

```
1: 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.*

**Notes:**

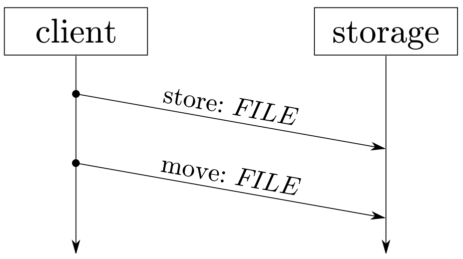

Figure 2.5: A simple protocol

**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 2.6** Client-Server Algorithm with Acknowledgments

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

**Notes:**

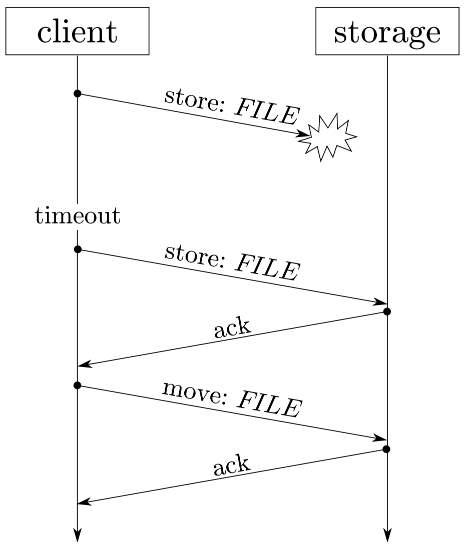

Figure 2.7: A protocol that handles message loss.

**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 2.8** (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 2.9.** *If Algorithm 2.6 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 $x = (0 \cdot 2) + 1 = 1$.

□

**Definition 2.10** (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 6 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 2.11** State Replication with a Serializer

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

**Remarks:**

- This idea is sometimes also referred to as *master-slave 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*.

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

**Algorithm 2.12** Two-Phase Protocol

```
   Phase 1
1: Client asks all servers for the lock
   Phase 2
2: if client receives lock from every server then
3:    Client sends command reliably to each server, and gives the lock back
4: else
5:    Clients gives the received locks back
6:    Client waits, and then starts with Phase 1 again
7: 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 2.12 really handle node crashes well? No! In fact, it is even worse than the simple serializer approach (Algorithm 2.11): Instead of needing one available node, Algorithm 2.12 requires *all* servers to be responsive!
- Does Algorithm 2.12 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 2.13** (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 2.12 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 2.14** 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? **Notes:** Even if a majority of the servers store $c_1$, a client $u_2$ might not realize that there is a majority, since $u_2$ is only required to contact *any* majority in Phase 1, and not all servers.
- 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 2.15** Paxos

```
Client (Proposer)
[1.9]Initialization
$c$          ◃ command to execute
$t = 0$      ◃ ticket number to try

Server (Acceptor)
$T_{\max} = 0$     ◃ largest issued ticket

$C = \perp$        ◃ stored command
$T_{\text{store}} = 0$   ◃ ticket used to store $C$

[1.9]Phase 1
1: $t = t + 1$
2: Ask all servers for ticket $t$

3: if $t > T_{\max}$ then
4:     $T_{\max} = t$
5:     Answer with ok($T_{\text{store}}$, $C$)
6: end if

[1.9]Phase 2
7: if a majority answers ok then
8:     Pick ($T_{\text{store}}$, $C$) with largest $T_{\text{store}}$
9:     if $T_{\text{store}} > 0$ then
10:        $c = C$
11:    end if
12:    Send propose($t$, $c$) to same majority
13: end if

14: if $t = T_{\max}$ then
15:     $C = c$
16:     $T_{\text{store}} = t$
17:     Answer success
18: end if

[1.9]Phase 3
19: if a majority answers success then
20:     Send execute($c$) to every server
21: 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 2.16.** *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 $\tau$ since clients only send a proposal if they received a majority of the tickets for $\tau$ (Line 7). Hence, every proposal is uniquely identified by its ticket number $\tau$.

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. Recall that since `propose`$(t,c)$ has been chosen, this means that 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 2.17.** *If a command c is executed by some servers, all servers (eventually) execute c.*

*Proof.* From Lemma 2.16 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. **Notes:** Of course, all proposers (clients) could crash before they succeed in informing all acceptors (servers). But in that case, there is no client anymore anyway. . . and if a new client ever starts to participate in the system, Paxos guarantees that this client will propose command $c$ to every acceptor.

- 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 2.10, 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

**Notes:**

- TODO: need to fix lemma 2.16: it must say somewhere before (or in title of lemma) that this is for $f < n/2$.
- just briefly two friends, then model: consensus, nodes, crash, correct, msgs correct, agreement, termination, ask for trivial protocol always 0, then validity, then asynchronous model (upon, no notion of time). then discuss a few possible protocols, suggested by students. Then claim that consensus cannot be solved with a single failure, and go into proof, as in script (configuration, univalent, 0,1-valent, bivalent, proof of bivalent initial configuration, transition, configuration tree, commutative transitions...) now break, which is not the best time.
- after the break continuing the proof (again path, critical, receiving nodes must be same for 0- and 1- (and then also for all 0- and 1-), and then let's crash that receiving node, and we are in a bivalent state. With about 35' on the clock in go into Ben-Or (blackboard), explaining algorithm, validity, then that only one value can be proposed, then the case where somebody manages to go into decision branch, and all others will go into second branch, and finally the case where nobody goes into decision branch, but all others luckily choose the same. Quick quip with replacing the randomized line with just deterministically choosing 0 (not too much, it is in the exercises). Finally (almost 20' on clock, I am hurrying already) shared coin: coins, sets, multiset C with $(n-f)^2$ coins. Let $W$ be all coins I have in at least $f+1$ sets. I claim that $|W| \geq f+1$. If not, we have strictly less than $(n-f)^2$ coins in multiset. Hence true, and also coins in $W$ are seen by everybody! Then quick (5') probability math for 1 and 0, and then plugging it back in. The second part today is stressful, it would be good to finish FLP proof in the first part to have enough time for second part.

## 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.

## 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 2.8).

**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 $V = [v_0, v_1, \ldots, v_{n-1}]$ denote the array of input values, where $v_i$ is the input value of node $i$.

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

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 $C_\tau$ 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 $C$.
- $C_\tau$ 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 $\tau_1$ and then $\tau_2$, 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 $\tau_2$ is applicable to $C_{\tau_1}$, since $m_2$ is still in transit and $\tau_1$ cannot change the state of $u_2$. With the same argument $\tau_1$ is applicable to $C_{\tau_2}$, 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 $C_{\tau_0}$ be 0-valent and $C_{\tau_1}$ 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 $C_{\tau_0}$ it must be 0-valent. However, this configuration also follows $C_{\tau_1}$ 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 (Ben-Or)

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

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 17). Note that once a node receives a majority of proposals for a value, it will adapt this value and terminate in the next 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 decides, 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 $r$. Therefore, all nodes will broadcast $v$ at the end of round $r$, and thus all nodes will propose $v$ in round $r+1$. The nodes that already decided in round $r$ will terminate in $r+1$ and send one additional `myValue` message (Line 13). All other nodes will receive a majority of proposals for $v$ in $r+1$, and will set decided to true in round $r+1$, and also send a `myValue` message in round $r+1$. Thus, in round $r+2$ some nodes have already terminated, and others hear enough `myValue` messages to continue in Line 6. They send another `propose` and a `myValue` message and terminate in $r+2$, deciding 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 two rounds 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 20). As shown before, all nodes that update the value based on a proposal, adapt the same value $v$. The rest of the nodes choses 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 $V_{\text{half}}$ all nodes in $N$ start with 0, and all nodes in $N'$ start with 1.

Assume that nodes start with $V_{\text{half}}$. 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 $V_{\text{half}}$. Analogously, the nodes in $N'$ cannot determine if they started in $V_1$ or $V_{\text{half}}$. 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. $\square$

**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 22?!
  **Notes:** This is a question for the audience
- 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. **Notes:**
  - This is covered in the exercise sheet, don't go into too many details.
  - $M$ is the majority, which all nodes have 0.
  - Only one node $m$ hears all messages from $M$.

- $m$ proposes 0, all other nodes propose $\perp$.
- All nodes in $M$ hear the proposal of $m$ and stay with 0. All other nodes (the minority) do not hear $m$ and choose deterministically 1.
- Repeat forever.

- 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 22 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.

**Notes:**

- we need a better shared coin, even only in the case of crash failures! darya is working on a new chapter...

**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]. 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.

[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

**Notes:**

- Quick recap from chapter 2, then story why crash failures may be wrong, and then byzantine/agreement. Then discussion about validity. After the first 20' derive $f = 1$ algo, and then with about 15' on clock $n = 4$ lower bound and then $3f + 1$ lower bound. At the end somebody asks for $3f + 1$ upper bound, and I quickly sketch the exponential paths algorithm. Break 2' early. (This year I didn't even quite manage to do the general $f \geq n/3$ proof, which I must study carefully!)

- After the break the King algorithm, with proof (maybe 20'), and then quickly the $f + 1$ round lower bound (the simple version, very quick. With about 20' on clock left into byzantine Ben-Or, including slow proof. Altogether about 5' before the end finished, time for discussion (in particular whether any of this can be considered practical). All in all a good lecture, even though I am finished 2' early twice.

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 arbitrary behavior before stopping completely. With these insights researchers modeled failures as arbitrary failures, not restricted to any patterns.

**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 which 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 deliever 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 deteced and discarded. It is worse if byzantine nodes send syntactically correct messages, but with a 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 say anything about 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. One can achieve all-same validity by choosing the smallest value that occurs at least twice, if a value appears 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 $n = 3$?

**Notes:**

- generalized algorithm sketch: memorize all paths without loops.

**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 of the 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
     Round 1
3:   Broadcast value($x$)
     Round 2
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
     Round 3
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 Round 2. 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 round 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 round. □

**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: r = 1                        ◁ round
3: decided = false
4: Broadcast propose($x_u$,r)
5: repeat
6:    Wait until $n - f$ propose messages of current round $r$ arrived
7:    if at least $n/2 + 3f + 1$ propose messages contain same value $x$ then
8:       $x_u = x$, decided = true
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:   r = r + 1
15:   Broadcast propose($x_u$,r)
16: until decided (see Line 8)
17: decision = $x_u$
```

**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 6, 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, the Ben-Or algorithm terminates within expected constant number of rounds for small values of $f$.

**Notes:**

**Remarks:**

- Since byzantine agreement seems to be hard, we might want to have an algorithm that learns as much information as possible. At the beginning, all nodes only know their initial value, hence, the most they can do is send around this value. In all later rounds, they can at most send around all information they learned in all previous rounds – but how can we structure this information?

**Definition 4.24** (value with path)**.** *We call $m$ a **value with path**, if $m = (u \to v \to \ldots \to w, x)$. The first argument of the message is the **path** that the* *message took (originated from $u$, sent to $v$, $\dots$, and at the end received by $w$), and the second argument $x$ is the* ***value****.*

**Remarks:**

- In the first round of an algorithm, every node $u$ with initial value $x$ can send $(\perp, x)$ to all other nodes.
- In the second round, every node $u$ already received $(\perp, y)$ from every node $v$. $u$ updates the path, since it received the message from $v$, such that the message becomes $(v, y)$ and sends it to all other nodes. In the third round, the message is received by a node $w$, which updates the message to $(v \to u, y)$.
- Every node always updates the path upon *receiving* a message; this prevents byzantine nodes from lying about the sender address.

**Algorithm 4.25** Exponential Information Gathering

```
$u$              ◁ own id
$x$              ◁ initial value
$R = \{\}$              ◁ set of received values with path
$V$ = new Array(n), each entry $\infty$        ◁ learned initial values from all nodes

Information gathering
1: $P = \{(\perp, x)\}$
2: for round = 1 to $f+1$ do
3:     Broadcast valuesWithPaths($P$)
4:     Receive values with path $P_v$ from every node $v$
5:     Remove all values with path from each $P_v$ with a path length $\neq$ round$-1$
6:     Add $v$ to all paths of all values with path in $P_v$
7:     $P = \bigcup_v P_v$
8:     Add all values with paths in $P$ to $R$
9: end for

Information analysis
10: Remove all values with path from $R$ which have a path containing $u$
11: Remove all values with path from $R$ which have a non-node-disjoint path
12: $V[u] = x$                    ◁ $u$ knows its own value
13: for all nodes $v \neq u$ do
14:     From$_v$ = all values with path in $R$ with a path starting from $v$
15:     if majority of tuples in From$_v$ have the same value $y$ then
16:         $V[v] = y$
17:     end if
18: end for
19: Decide on minimal value in $V$
```

**Remarks:**

- The goal of Algorithm 4.9 is that all nodes learn the same $V$, i.e., they agree on an initial value for each node.
- When they agree on all initial values, it is trivial to achieve consensus; e.g., they can simply pick the smallest initial value.
- We prove the correctness of this algorithm for $n \geq 4$ and $f = 1$.
- Observe that since $f = 1$, all nodes receive all values with path with path length 1 and 2.

**Lemma 4.26.** *Every node $u$ stores at least $1 + (n - f - 2)$ many values with path in $R$ for any correct node $v \neq u$, which all contain the true initial value $y$ of $v$.*

*Proof.* In Round 1 $u$ receives a value from $v$ directly, and puts $(v,\ y)$ into $R$. In Round 2, $u$ receives messages $(v,\ y)$ from all (at least $n - f$) correct nodes. Note that exactly two of these messages will be removed: One in Line 10, i.e., $(v \to u, y)$ and one in Line 11, i.e., $(v \to v, y)$.

Note that all these messages are only sent by correct nodes which do not alter the value, hence all these messages contain value $y$. □

**Lemma 4.27.** *All correct nodes learn the true initial value of all correct nodes.*

*Proof.* Since $f = 1$ and $n \geq 4$, every node stores at least $1 + (4 - 1 - 2) = 2$ tuples with the correct value in $R$. Let $b$ be the byzantine node. For every correct node $v$, it can only add a single value with path into $R$, namely $(v \to b,\ z)$. Thus, there are always at least two tuples for the true value, and at most one for a wrong value, hence the true value will be chosen. □

**Remarks:**

- Indeed the byzantine node could send more more paths in one round or not stick to the protocol at all. But such behavior can easily be detected, and the byzantine messages could simply be removed. We omitted such filtering of messages to enhance readability of the algorithm.

**Lemma 4.28.** *If any correct node $u$ choses a value $V[b] = y \neq \infty$ for a byzantine node $b$, every correct node $v$ choses the same $V[b] = y$.*

*Proof.* Recall that all values with path from $R$ which contain $b$ multiple times in the path are removed. Thus, for every correct node $u$, there is at most one message in $R$ which is directly received from $b$, namely $(b, *)$. Note that such a message must be sent in Round 1, as otherwise it would be removed (Line 5), as nodes could recognize it as incorrect behavior. Assume that a correct node $u$ received $(b, y)$ in Round 1. In that case, $u$ will relay this message, and all correct nodes will have $(b \to u, y)$ in their set $R$. Therefore, all nodes will have the same number of values with path for any value $y$ sent by $b$ in $R$. Since they all decide deterministically, all will decide for the same initial value of $b$. □

**Theorem 4.29.** *Algorithm 4.9 reaches byzantine agreement if $n \geq 4$.*

*Proof.* We need to show agreement, any-input validity and termination. With Lemmas 4.27 and 4.28 we know that all correct nodes have the same array $V$, and therefore agree on the same minimum value. The nodes agree on a value proposed by a node, so any-input validity holds. Moreover, the algorithm terminates after two rounds, if $f = 1$. □

**Remarks:**

- The algorithm works correctly for every $f \geq 0$ and $n \geq 3f + 1$, but the proof for a general $n$ and $f$ is significantly more involved.
- The disadvantage of this approach is that the message size is exponentially large in $n$, as there are a lot of paths between two nodes.
- Does Algorithm 4.9 also work with $n = 3$?

## 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. Is there a fast asynchronous algorithm, possibly based on some advanced communication methods?

## 5.1 Random Oracle and Bitstring

**Definition 5.1** (Random Oracle). *A random oracle is a trusted (non-byzantine) random source which can generate random values.*

**Algorithm 5.2** Shared Coin with Magic Random Oracle

```
1: return $c_i$, where $c_i$ is $i$th random bit by oracle
```

**Remarks:**

- Algorithm 5.2 as well as the following shared coin algorithms will for instance 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 throw a *shared* coin. In other words, the value $x_u$ in Line 12 of Algorithm 4.21 will be set to the return value of the shared coin subroutine.

- We have already seen a shared coin in Algorithm 3.22. This concept deserves a proper definition.

**Definition 5.3** (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.*

**Theorem 5.4.** *Algorithm 5.2 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 shared 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 5.2. 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 5.5** (Random Bitstring)**.** *A* ***random bitstring*** *is a string of random binary values, known to all participating nodes when starting a protocol.*

**Algorithm 5.6** Naive Shared Coin with Random Bitstring

```
1: 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 5.7.** *If the scheduling is worst-case, Algorithm 5.6 plugged into Algorithm 4.21 does not terminate.*

*Proof.* We start Algorithm 5.6 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 5.7 shows that a worst-case scheduler cannot be allowed to know the random bits of the future.
- Note that in the proof of Theorem 5.7 we did not even use any byzantine nodes. Just bad scheduling was enough to prevent termination.
- Worst-case scheduling is an issue that we have not considered so far, in particular in Chapter 3 we implicitly assumed that message scheduling was random. What if scheduling is worst-case in Algorithm 3.22?

**Lemma 5.8.** *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. □

**Remarks:**

- With worst-case asynchrony, some of our previous results do not hold anymore. Can we at least solve asynchronous (assuming worst-case scheduling) *consensus* if we have crash failures?
- This is indeed possible, but we need to sharpen our tools first.

## 5.2 Shared Coin on a Blackboard

**Definition 5.9** (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 5.10** Crash-Resilient Shared Coin with Blackboard (for node $u$)

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

**Remarks:**

- In Algorithm 5.10 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 5.10, 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 5.11** (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 real number $z$,*

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

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

**Theorem 5.12.** *Algorithm 5.10 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 $n^2 + n$ 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 5.3. 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 5.11 with $N = n^2$ and $z = 1$, we get:

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

□

**Lemma 5.13.** *Algorithm 5.10 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 5.11. 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 5.12, 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 5.10 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 5.10, we assume that the blackboard is a trusted central authority. Like the random oracle of Definition 5.1, 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.

## 5.3 Broadcast Abstractions

**Definition 5.14** (Accept)**.** *A message received by a node $v$ is called* ***accepted*** *if node $v$ can consider this message for its computation.*

**Definition 5.15** (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 5.16** (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 5.17** Asynchronous Reliable Broadcast (code for node $u$)

```
1: Broadcast own message msg($u$)
2: if received msg($v$) from node $v$ then
3:     Broadcast echo($u$, msg($v$))
4: end if
5: if received echo($w$, msg($v$)) from $n-2f$ nodes $w$ but not msg($v$) then
6:     Broadcast echo($u$, msg($v$))
7: end if
8: if received echo($w$, msg($v$)) from $n-f$ nodes $w$ then
9:     Accept(msg($v$))
10: end if
```

**Theorem 5.18.** *Algorithm 5.17 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*

*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 follows from byzantine nodes being unable to forge an incorrect sender address, see Definition 4.1.

The third property deals with a byzantine originator $b$. If a correct node accepted message msg($b$), this node must have received at least $n - f$ echoes for this message in Line 8. Since at most $f$ nodes are byzantine, at least $n - 2f$ correct nodes have broadcast an echo message for msg($b$). Therefore, every correct node will receive these $n - 2f$ echoes eventually and will broadcast an echo itself. Thus, all $n - f$ correct nodes will have broadcast an echo for msg($b$) and every correct node will accept msg($b$).

□

**Remarks:**

- Algorithm 5.17 does not terminate. Only *eventually*, all messages by correct nodes will be accepted.
- 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 5.19** (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 5.20** FIFO Reliable Broadcast (code for node $u$)

```
1: Broadcast own round $r$ message msg($u, r$)
2: if received first message msg($v, r$) from node $v$ for round $r$ then
3:    Broadcast echo($u$, msg($v, r$))
4: end if
5: if not echoed any msg'($v, r$) before then
6:    if received echo($w$, msg($v, r$)) from $f+1$ nodes $w$ but not msg($v, r$) then
7:       Broadcast echo($u$, msg($v, r$))
8:    end if
9: end if
10: if received echo($w$, msg($v, r$)) from $n-f$ nodes $w$ then
11:    if accepted msg($v, r-1$) then
12:       Accept(msg($v, r$))
13:    end if
14: end if
```

**Theorem 5.21.** *Algorithm 5.20 satisfies the properties of Theorem 5.18. Additionally, Algorithm 5.20 makes sure that no two messages* msg($v, r$) *and* msg'($v, r$) *are accepted from the same node. It can tolerate $f < n/3$ Byzantine nodes or $f < n/2$ crash failures.*

*Proof.* Just as reliable broadcast, Algorithm 5.20 satisfies the first two properties of Theorem 5.18 by simply following the flow of messages of a correct node.

For the third property, assume again that some message originated from a byzantine node $b$. If a correct node accepted message msg($b$), this node must have received at least $n - f$ echoes for this message in Line 10.

- Byzantine case: If at most $f$ nodes are byzantine, at least $n - 2f > f + 1$ correct nodes have broadcast an echo message for msg($b$).
- Crash-failure case: If at most $f$ nodes can crash, at least $n - f > f + 1$ nodes have broadcast an echo message for msg($b$).

In both cases, every correct node will receive these $f+1$ echoes eventually and will broadcast an echo. Thus, all $n-f$ correct nodes will have broadcast an echo for $\texttt{msg}(b)$ and every correct node will accept $\texttt{msg}(b)$.

It remains to show that at most one message will be accepted from some node $v$ in a round $r$.

- Byzantine case: Assume that some correct node $u$ has accepted $\texttt{msg}(v,r)$ in Line 12. Then, $u$ has received $n-f$ echoes for this message, $n-2f$ of which were the first echoes of the correct nodes. Assume for contradiction that another correct node accepts $\texttt{msg'}(v,r)$. This node must have collected $n-f$ messages echo$(w, \texttt{msg'}(v,r))$. Since at least $n-2f$ of these messages must be the first echo messages sent by correct nodes, we have $n-2f+n-2f=2n-4f>n-f$ (for $f<n/3$) echo messages sent by the correct nodes as their first echo. This is a contradiction.

- Crash-failure case: At least $n-2f$ not crashed nodes must have echoed $\texttt{msg}(v,r)$, while $n-f$ nodes have echoed $\texttt{msg'}(v,r)$. In total $2n-3f>n-f$ (for $f<n/2$) correct nodes must have echoed either of the messages, which is a contradiction.

□

**Definition 5.22** (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 5.22 is equivalent to Definition 2.10, i.e., atomic broadcast = state replication.

- Now we have all the tools to finally solve asynchronous consensus.

**Notes:**

## 5.4 Blackboard with Message Passing

**Algorithm 5.23** Crash-Resilient Asynchronous Shared Coin (code for node $u$)

```
1: In general, node $u$ will only participate in (echo) FIFO-broadcasting
   coin($i+1, c_w$) if node $u$ already accepted the previous coin($i, c_w$)

   Phase 1: Build bb-matrix row by row

2: for round $j = 1$ to $n$ do
3:    Choose local $c_u = +1$ with probability $1/2$, else $c_u = -1$
4:    FIFO-broadcast coin($j, c_u$)
5:    Wait until accepted coin($j, c_w$) from $n - f$ nodes
6: end for
7: Save all accepted messages as a matrix bb($u$)

   Phase 2: Update bb-matrix

8: FIFO-broadcast bb($u$)
9: repeat
10:   Update accepted entries in bb($u$)
11:   Echo bb($w$) only if bb($w$) $\subseteq$ bb($u$) and bb($w$) has $n - f$ full columns
12: until accepted $(f + 1)$ bb-matrices

   Phase 3: Decide on the coinflip

13: Broadcast updated matrix bb($u$)
14: Wait for $n - f$ other updated matrices
15: Update every entry in bb($u$) that was inside of at least one bb-matrix
16: return sign(sum(bb($u$)))
```

**Lemma 5.24.** *At the end of Phase 1, matrix* `bb`$(u)$ *will contain* $n-f$ *columns, each having* $n$ *accepted coinflips.*

*Proof.* The matrix `bb`$(u)$ only contains accepted entries at the end of Phase 1. Each row of the matrix represents a round of communication and each column represents a node. A node only increments its round when it has accepted all coinflips from the previous rounds from $n - f$ different nodes (including itself). After the last round, the `bb`$(u)$ matrix holds $n - f$ full columns and $f$ columns where the top part is filled and the rest of the values is unknown. □

**Lemma 5.25.** *All nodes will finish Phase 2 of Algorithm 5.23.*

*Proof.* Due to Theorem 5.18(3), every value of any correct `bb` matrix will be accepted by every correct node eventually. Since the nodes keep updating their `bb` matrix in Line 10, all correct nodes will eventually contain all correct matrices as submatrices and participate in their FIFO broadcast. Once the first correct node accepts $(t + 1)$ matrices, all nodes will accept the matrices eventually, an therefore Phase 2 of the algorithm will be completed eventually. □

**Lemma 5.26.** *At the end of Phase 3, all correct* `bb` *matrices share the same* $(n - f) \times (n)$ *submatrix.*

*Proof.* In order to show that there will be a common submatrix, we need consider Phase 2 again. There, each node accepts $f+1$ blackboard matrices. Among these matrices there will be a matrix `bb`$(v)$ from a correct node. Since the matrices were FIFO broadcast, at least $n-2f$ correct nodes have accepted all values from `bb`$(v)$ and will broadcast a matrix containing these values in Line 13. Every node also receives matrices from at least $n-f$ correct nodes, $n-2f$ of which will contain all values from matrix `bb`$(v)$. The above statement follows from the assumption that $f < n/2$ and Lemma 5.24. □

**Lemma 5.27.** *At the end of the algorithm, the remaining $f$ columns will only differ in the latest value a node has received.*

*Proof.* Assume one node has accepted a value in the round $i$. Then at least $n-2f$ correct nodes have participated in FIFO broadcast of this value. By the condition in Line **??** these correct nodes must have accepted the value from the same node in the previous round. This means, at least $n-2f$ correct nodes have accepted the value from the previous round. Therefore, every correct node will accept this value in Phase 3 of the algorithm. □

**Theorem 5.28.** *Algorithm 5.23 solves the asynchronous agreement problem with crash failures in the message passing model with high probability, while exchanging $O(n^4)$ messages.*

*Proof.* The blackboard model makes sure that for any number of iterations chosen in Phase 1 of the algorithm, the local views of nodes differ in the same $f$ entries. For $n$ rounds, the total number of generated coinflips is $n^2$, just as in Algorithm 5.10. With only $f$ different entries, the analysis from Theorem 5.12 can be applied here. This implies a shared coin which generates the values $-1$ and 1 with constant probability.

The largest number of messages that is exchanged in the algorithm, is exchanged in Phase 2. There, each `bb` matrix which contains $n^2$ messages is broadcast reliably. This results in $O(n^4)$ messages in total. □

**Remarks:**

- The probability of success can be increased by either increasing the number of rounds (i.e. total number of coinflips) in Phase 1 of the algorithm or executing the algorithm repeatedly.

- Algorithm 5.23 can be modified to tolerate $n/3$ byzantine nodes, while satisfying the same conditions as in the crash failure model.

- Even though the assumption of only $f$ different values among $n^2$ or even $n^c$ coinflips is very appealing in the byzantine setting, also this method has exponential expected running time. This is because byzantine nodes control $f \cdot n \approx n^2$ entries of the matrix, and can thus change the deviation of the sum easily.

## 5.5 Blackboard with Message Passing

**Algorithm 5.29** Crash-Resilient Shared Coin (code for node $u$)

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

**Theorem 5.30.** *Algorithm 5.29 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 5.21. The idea of this algorithm is to simulate the read and write operations from Algorithm 5.10.

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 `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 `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 5.10. 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 5.12 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!

## 5.6 Using Cryptography

**Definition 5.31** (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 5.32** (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 5.33** $(t, n)$-Threshold Secret Sharing

```
1: Input: A secret $s$, represented as a real number.
   Secret distribution by dealer $d$
2: Generate $t-1$ random numbers $a_1, \dots, a_{t-1} \in \mathbb{R}$
3: Obtain a polynomial $p$ of degree $t-1$ with $p(x) = s + a_1x + \dots + a_{t-1}x^{t-1}$
4: Generate $n$ distinct $x_1, \dots, x_n \in \mathbb{R} \setminus \{0\}$
5: 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
6: Collect $t$ shares $\texttt{msg}(x_u, p(x_u))_d$ from at least $t$ nodes
7: Use Lagrange's interpolation formula to obtain $p(0) = s$
```

**Remarks:**

- Algorithm 5.33 relies on a trusted dealer, who broadcasts the secret shares to the 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 5.34** Preprocessing Step for Algorithm 5.35 (code for dealer $d$)

```
1: According to Algorithm 5.33, 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 5.33, 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 5.35** Shared Coin using Secret Sharing ($i$th iteration)

```
1: Request shares from at least $f + 1$ nodes
2: Using Algorithm 5.33, let $c_i$ be the value reconstructed from the shares
3: return $c_i$
```

**Theorem 5.36.** *Algorithm 4.21 together with Algorithm 5.34 and Algorithm 5.35 solves asynchronous byzantine agreement for $f < n/3$ in expected* 3 *number of rounds.*

*Proof.* In Line 1 of Algorithm 5.35, 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 2 of the algorithm. The running time analysis follows then from the analysis of Theorem 5.4. □

**Remarks:**

- In Algorithm 5.34 we assume that the dealer generates the random bitstring. This assumption is not necessary in general.
- 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 5.37** (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 5.38** 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 $\mathtt{msg}(r)_u$, i.e., round number $r$ signed by node $u$
4: Compute $h_v = \text{hash}(\text{msg}(r)_v)$ for all received messages $\mathtt{msg}(r)_v$
5: Let $h_{min} = \min_v \ h_v$
6: return least significant bit of $h_{min}$
```

**Remarks:**

- In Algorithm 5.38, 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 5.39.** *Algorithm 5.38 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. The expected number of rounds is:

$$1 + \sum_{i=0}^{\infty} 2 \cdot \left(\frac{2}{3}\right)^i = 5$$

□

## 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 5.10 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 Algorithm 5.29. 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 5.34 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 5.38 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.