Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Consensus

Roger Wattenhofer

wattenhofer@ethz.ch

Summer School May-June 2016

Contents

Chapter 1

Fault-Tolerance & Paxos

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

1.1 Client/Server

Definition 1.1 (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 1.2 (message passing)**.** *In the* ***message passing model*** *we study distributed systems that consist of a set of nodes. Each node can perform local computations, and can send messages to every other node.*

Remarks:

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

Algorithm 1.3 Naïve Client-Server Algorithm

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

Model 1.4 (message loss)**.** *In the message passing model with* ***message loss****, for* ***any*** *specific message, it is not guaranteed that it will arrive safely at the receiver.*

Remarks:

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

- Algorithm 1.3 does not work correctly if there is message loss, so we need a little improvement.

Algorithm 1.5 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
```

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

Remarks:

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

Theorem 1.7. *If Algorithm 1.5 is used with multiple clients and multiple servers, the servers might see the commands in different order, leading to an inconsistent state.*

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

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

□

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

Definition 1.8 (state replication)**.** *A set of nodes achieves* ***state replication,*** *if all nodes execute a (potentially infinite) sequence of commands $c_1, c_2, c_3, \dots,$ 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 **??** 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 1.9 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*.

Algorithm 1.10 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 1.10 really handle node crashes well? No! In fact, it is even worse than the simple serializer approach (Algorithm 1.9): Instead of having a only one node which must be available, Algorithm 1.10 requires *all* servers to be responsive!
- Does Algorithm 1.10 also work if we only get the lock from a subset of servers? Is a majority of servers enough?
- What if two or more clients concurrently try to acquire a majority of locks? Do clients have to abandon their already acquired locks, in order not to run into a deadlock? How? And what if they crash before they can release the locks? Do we need a slightly different concept?

1.2 Paxos

Definition 1.11 (ticket)**.** *A* ***ticket*** *is a weaker form of a lock, with the following properties:*

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

Remarks:

- There is no problem with crashes: If a client crashes while holding a ticket, the remaining clients are not affected, as servers can simply issue new tickets.
- Tickets can be implemented with a counter: Each time a ticket is requested, the counter is increased. When a client tries to use a ticket, the server can determine if the ticket is expired.
- What can we do with tickets? Can we simply replace the locks in Algorithm 1.10 with tickets? We need to add at least one additional phase, as only the client knows if a majority of the tickets have been valid in Phase 2.

Algorithm 1.12 Naïve Ticket Protocol

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

Remarks:

- There are problems with this algorithm: Let u_1 be the first client that successfully stores its command c_1 on a majority of the servers. Assume that u_1 becomes very slow just before it can notify the servers (Line 7), and a client u_2 updates the stored command in some servers to c_2. Afterwards, u_1 tells the servers to execute the command. Now some servers will execute c_1 and others c_2!

- How can this problem be fixed? We know that every client u_2 that updates the stored command after u_1 must have used a newer ticket than u_1. As u_1's ticket was accepted in Phase 2, it follows that u_2 must have acquired its ticket after u_1 already stored its value in the respective server.

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

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

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

- So, in order to determine which command was stored most recently, servers can remember the ticket number that was used to store the command, and afterwards tell this number to clients in Phase 1.
- If every server uses its own ticket numbers, the newest ticket does not necessarily have the largest number. This problem can be solved if clients suggest the ticket numbers themselves!

Algorithm 1.13 Paxos

```
Client (Proposer)                          Server (Acceptor)

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

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

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

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

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

Assume that there is at least one `propose`(t',c') with $t' > t$ and $c' \neq c$; of such proposals, consider the proposal with the smallest ticket number t'. Since both this proposal and also the `propose`(t,c) have been sent to a majority of the servers, we can denote by S the non-empty intersection of servers that have been involved in both proposals. 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 $t^* > t$. But in that 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 1.15. *If a command c is executed by some servers, all servers (eventually) execute c.*

Proof. From Lemma 1.14 we know that once a proposal for c is chosen, every subsequent proposal is for c. As there is exactly one first `propose`(t,c) that is chosen, it follows that all successful proposals will be for the command c. Thus, only proposals for a single command c can be chosen, and since clients only tell servers to execute a command, when it is chosen (Line 20), each client will eventually tell every server to execute c. □

Remarks:

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

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

- Note that Paxos cannot make progress if half (or more) of the servers crash, as clients cannot achieve a majority anymore.

- The original description of Paxos uses three roles: Proposers, acceptors and learners. Learners have a trivial role: They do nothing, they just learn from other nodes which command was chosen.

- We assigned every node only one role. In some scenarios, it might be useful to allow a node to have multiple roles. For example in a peer-to-peer scenario nodes need to act as both client and server.

- Clients (Proposers) must be trusted to follow the protocol strictly. However, this is in many scenarios not a reasonable assumption. In such scenarios, the role of the proposer can be executed by a set of servers, and clients need to contact proposers, to propose values in their name.

- So far, we only discussed how a set of nodes can reach decision for a single command with the help of Paxos. We call such a single decision an *instance* of Paxos.

- If we want to execute multiple commands, we can extend each instance with an instance number, that is sent around with every message. Once a command is chosen, any client can decide to start a new instance with the next number. If a server did not realize that the previous instance 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].

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.

Chapter 2

Consensus

2.1 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?

2.2 Consensus

In Chapter 1 we studied a problem that we vaguely called agreement. We will now introduce a formally specified variant of this problem, called *consensus*.

Definition 2.1 (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.
- 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.

2.3 Impossibility of Consensus

Model 2.2 (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 1.6).

Definition 2.3 (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 2.4 (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 2.5 (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 2.6 (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 sent some messages, and are now waiting for the first message to arrive.

Lemma 2.7. *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. Note that C_0 only depends on the input values of the nodes, as no event occurred yet. 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 is 0-valent, and configuration corresponding to V_{b+1} is 1-valent. Observe that only the input value of the b^{th} node differs from V_b to V_{b+1}.

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 respectively V_{b+1}. 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 is 0-valent, v must be 0. However we know that V_{b+1} is 1-valent, thus v must be 1. Since v cannot be both 0 and 1, we have a contradiction.

□

Definition 2.8 (transition)**.** *A **transition** from configuration C to a following configuration C_τ is characterized by an event $\tau = (u, m)$, i.e., node u receiving message m.*

Remarks:

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

Definition 2.9 (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 2.10. *Assume two transitions $\tau_1 = (u_1, m_1)$ and $\tau_2 = (u_2, m_2)$ for $u_1 \neq u_2$ are both applicable to C. Let $C_{\tau_1\tau_2}$ be the configuration that follows C by first applying transition τ_1 and then τ_2, and let $C_{\tau_2\tau_1}$ be defined analogously. It holds that $C_{\tau_1\tau_2} = C_{\tau_2\tau_1}$.*

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

Definition 2.11 (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 2.12. *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 2.7). 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 2.13. *If a configuration tree contains a critical configuration, crashing a single node can create a bivalent leaf; i.e., a crash prevents the algorithm from reaching agreement.*

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

Assume that $u_0 \neq u_1$. Using Lemma 2.10 we know that C has a following configuration $C_{\tau_0\tau_1} = C_{\tau_1\tau_0}$. Since this configuration follows C_{τ_0} it must be 0-valent. However, this configuration also follows C_{τ_1} and must hence be 1-valent. This is a contradiction and therefore $u_0 = u_1$ must hold.

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

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

□

Theorem 2.14. *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 2.7 we know that there must be at least one bivalent initial configuration C. Using Lemma 2.12 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 2.13). □

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.

2.4 Randomized Consensus

Algorithm 2.15 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
     Adapt
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:    decide = 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 2.15 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 2.16. *As long as no node sets* ***decided*** *to true, Algorithm 2.15 always makes progress, independent of which nodes crash.*

Proof. The only two steps in the algorithm when a node waits are in Lines 6 and 15. 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 2.17. *Algorithm 2.15 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 2.18. *Algorithm 2.15 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 2.16 we only showed that nodes make progress 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 make progress in Line 6. They send another `propose` and a `myValue` message and terminate in $r+2$, deciding for the same value v. □

Lemma 2.19. *Algorithm 2.15 satisfies the termination requirement, i.e., all nodes terminate in expected time $O(2^n)$.*

Proof. We know from the proof of Lemma 2.18 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 2.20. *Algorithm 2.15 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 2.21. *There is no consensus algorithm for the asynchronous model that tolerates $f \geq n/2$ many failures.*

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

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

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

□

Remarks:

- Algorithm 2.15 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?!

- This cannot work: Such a change makes the algorithm deterministic, and therefore it cannot achieve consensus (Theorem 2.14). Simulating what happens by always choosing 1, one can see that it might happen that there is a majority for 0, but a minority with value 1 prevents the nodes from reaching agreement.

- Nevertheless, the algorithm can be improved by tossing a so-called *shared coin*. A shared coin is a random variable that is 0 for all nodes with constant probability, and 1 with constant probability. Of course, such a coin is not a magic device, but it is simply an algorithm. To improve the expected runtime of Algorithm 2.15, we replace Line 22 with a function call to the shared coin algorithm.

2.5 Shared Coin

Algorithm 2.22 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 2.23. *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 number of total 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 2.24. *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 2.25. *If $f < n/3$ nodes crash, Algorithm 2.22 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 2.23 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 2.24), and hence everybody will decide 0. Thus Algorithm 2.22 implements a shared coin. $\square$

Remarks:

- We only proved the worst case. By choosing f fairly small, it is clear that $f + 1 \not\approx n/3$. However, Lemma 2.23 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 2.25 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.

Theorem 2.26. *Plugging Algorithm 2.22 into Algorithm 2.15 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].

This chapter was written in collaboration with David Stolz.

Bibliography

[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.

[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 3

Authenticated Agreement

Byzantine nodes are able to lie about their inputs as well as received messages. Can we detect certain lies and limit the power of byzantine nodes? Possibly, the authenticity of messages may be validated using signatures?

3.1 Agreement with Authentication

Definition 3.1 (Signature)**.** *If a node never signs a message, then no correct node ever accepts that message. We denote a message* $\texttt{msg}(x)$ *signed by node* u *with* $\texttt{msg}(x)_u$*.*

Remarks:

- Algorithm 3.2 shows an agreement protocol for binary inputs relying on signatures. We assume there is a designated "primary" node p. The goal is to decide on p's value.

Algorithm 3.2 Byzantine Agreement with Authentication

```
  Code for primary p:
if input is 1 then
  broadcast $\texttt{value}(1)_p$
  decide 1 and terminate
else
  decide 0 and terminate
end if
  Code for all other nodes v:
for all rounds $i \in 1, \dots, f+1$ do
  $S$ is the set of accepted messages $\texttt{value}(1)_u$.
  if $|S| \geq i$ and $\texttt{value}(1)_p \in S$ then
    broadcast $S \cup \{\texttt{value}(1)_v\}$
    decide 1 and terminate
  end if
end for
decide 0 and terminate
```

Theorem 3.3. *Algorithm 3.2 can tolerate $f < n$ byzantine failures while terminating in $f + 1$ rounds.*

Proof. Assuming that the primary p is not byzantine and its input is 1, then p broadcasts $\texttt{value}(1)_p$ in the first round, which will trigger all correct nodes to decide for 1. If p's input is 0, there is no signed message $\texttt{value}(1)_p$, and no node can decide for 1.

If primary p is byzantine, we need all correct nodes to decide for the same value for the algorithm to be correct. Let us assume that p convinces a correct node v that its value is 1 in round i with $i < f + 1$. We know that v received i signed messages for value 1. Then, v will broadcast $i + 1$ signed messages for value 1, which will trigger all correct nodes to also decide for 1. If p tries to convince some node v late (in round $i = f + 1$), v must receive $f + 1$ signed messages. Since at most f nodes are byzantine, at least one correct node u signed a message $\texttt{value}(1)_u$ in some round $i < f + 1$, which puts us back to the previous case. □

Remarks:

- The algorithm only takes $f + 1$ rounds, which is optimal as described in Theorem **??**.
- Using signatures, Algorithm 3.2 solves consensus for any number of failures! Does this contradict Theorem **??**? Recall that in the proof of Theorem **??** we assumed that a byzantine node can distribute contradictory information about its own input. If messages are signed, correct nodes can detect such behavior – a node u signing two contradicting messages proves to all nodes that node u is byzantine.
- Does Algorithm 3.2 satisfy any of the validity conditions introduced in Section **??**? No! A byzantine primary can dictate the decision value. Can we modify the algorithm such that the correct-input validity condition is satisfied? Yes! We can run the algorithm in parallel for $2f + 1$ primary nodes. Either 0 or 1 will occur at least $f + 1$ times, which means that one correct process had to have this value in the first place. In this case, we can only handle $f < \frac{n}{2}$ byzantine nodes.
- In reality, a primary will usually be correct. If so, Algorithm 3.2 only needs two rounds! Can we make it work with arbitrary inputs? Also, relying on synchrony limits the practicality of the protocol. What if messages can be lost or the system is asynchronous?
- Zyzzyva uses authenticated messages to achieve state replication, as in Definition 1.8. It is designed to run fast when nodes run correctly, and it will slow down to fix failures!

3.2 Zyzzyva

Definition 3.4 (View)**.** *A view V describes the current state of a replicated system, enumerating the $3f + 1$ replicas. The view V also marks one of the replicas as the primary p.*

Definition 3.5 (Command). *If a client wants to update (or read) data, it sends a suitable command c in a* `Request` *message to the primary p. Apart from the command c itself, the* `Request` *message also includes a timestamp t. The client signs the message to guarantee authenticity.*

Definition 3.6 (History). *The history h is a sequence of commands $c_1, c_2, \ldots$ in the order they are executed by Zyzzyva. We denote the history up to c_k with h_k.*

Remarks:

- In Zyzzyva, the primary p is used to order commands submitted by clients to create a history h.
- Apart from the globally accepted history, node u may also have a local history, which we denote as h^u or h_k^u.

Definition 3.7 (Complete command). *If a command completes, it will remain in its place in the history h even in the presence of failures.*

Remarks:

- As long as clients wait for the completion of their commands, clients can treat Zyzzyva like one single computer even if there are up to f failures.

In the Absence of Failures

Algorithm 3.8 Zyzzyva: No failures

```
1: At time $t$ client $u$ wants to execute command $c$
2: Client $u$ sends request R = Request$(c,t)_u$ to primary $p$
3: Primary $p$ appends $c$ to its local history, i.e., $h^p = (h^p, c)$
4: Primary $p$ sends OR = OrderedRequest$(h^p, c,$ R$)_p$ to all replicas
5: Each replica $r$ appends command $c$ to local history $h^r = (h^r, c)$ and checks
   whether $h^r = h^p$
6: Each replica $r$ runs command $c_k$ and obtains result $a$
7: Each replica $r$ sends Response$(a,$OR$)_r$ to client $u$
8: Client $u$ collects the set $S$ of received Response$(a,$OR$)_r$ messages
9: Client $u$ checks if all histories $h^r$ are consistent
10: if $|S| = 3f + 1$ then
11:     Client $u$ considers command $c$ to be complete
12: end if
```

Remarks:

- Since the client receives $3f+1$ consistent responses, all correct replicas have to be in the same state.
- Only three communication rounds are required for the command c to complete.

- Note that replicas have no idea which commands are considered complete by clients! How can we make sure that commands that are considered complete by a client are actually executed? We will see in Theorem 3.23.
- Commands received from clients should be ordered according to timestamps to preserve the causal order of commands.
- There is a lot of optimization potential. For example, including the entire command history in most messages introduces prohibitively large overhead. Rather, old parts of the history that are agreed upon can be truncated. Also, sending a hash value of the remainder of the history is enough to check its consistency across replicas.
- What if a client does not receive $3f + 1$ `Response`$(a$,`OR`$)_r$ messages? A byzantine replica may omit sending anything at all! In practice, clients set a timeout for the collection of `Response` messages. Does this mean that Zyzzyva only works in the synchronous model? Yes and no. We will discuss this in Lemma 3.26 and Lemma 3.27.

Byzantine Replicas

Algorithm 3.9 Zyzzyva: Byzantine Replicas (append to Algorithm 3.8)

```
1: if $2f + 1 \leq |S| < 3f + 1$ then
2:     Client $u$ sends Commit$(S)_u$ to all replicas
3:     Each replica $r$ replies with a LocalCommit$(S)_r$ message to $u$
4:     Client $u$ collects at least $2f + 1$ LocalCommit$(S)_r$ messages and considers
       $c$ to be complete
5: end if
```

Remarks:

- If replicas fail, a client u may receive less than $3f + 1$ consistent responses from the replicas. Client u can only assume command c to be complete if all correct replicas r eventually append command c to their local history h^r.

Definition 3.10 (Commit Certificate)**.** *A commit certificate S contains $2f + 1$ consistent and signed* `Response`$(a$,`OR`$)_r$ *messages from $2f + 1$ different replicas r.*

Remarks:

- The set S is a commit certificate which proves the execution of the command on $2f + 1$ replicas, of which at least $f + 1$ are correct. This commit certificate S must be acknowledged by $2f + 1$ replicas before the client considers the command to be complete.
- Why do clients have to distribute this commit certificate to $2f + 1$ replicas? We will discuss this in Theorem 3.21.

- What if $|S| < 2f + 1$, or what if the client receives $2f + 1$ messages but some have inconsistent histories? Since at most f replicas are byzantine, the primary itself must be byzantine! Can we resolve this?

Byzantine Primary

Definition 3.11 (Proof of Misbehavior)**.** *Proof of misbehavior of some node can be established by a set of contradicting signed messages.*

Remarks:

- For example, if a client u receives two `Response`$(a,$`OR`$)_r$ messages that contain inconsistent `OR` messages signed by the primary, client u can prove that the primary misbehaved. Client u broadcasts this proof of misbehavior to all replicas r which initiate a view change by broadcasting a `IHatePrimary`$_r$ message to all replicas.

Algorithm 3.12 Zyzzyva: Byzantine Primary (append to Algorithm 3.9)

```
1: if $|S| < 2f + 1$ then
2:   Client $u$ sends the original R = Request$(c,t)_u$ to all replicas
3:   Each replica $r$ sends a ConfirmRequest(R)$_r$ message to $p$
4:   if primary $p$ replies with OR then
5:     Replica $r$ forwards OR to all replicas
6:     Continue as in Algorithm 3.8, Line 5
7:   else
8:     Replica $r$ initiates view change by broadcasting IHatePrimary$_r$ to all
       replicas
9:   end if
10: end if
```

Remarks:

- A faulty primary can slow down Zyzzyva by not sending out the `OrderedRequest` messages in Algorithm 3.8, repeatedly escalating to Algorithm 3.12.
- Line 5 in the Algorithm is necessary to ensure liveness. We will discuss this in Theorem 3.27.
- Again, there is potential for optimization. For example, a replica might already know about a command that is requested by a client. In that case, it can answer without asking the primary. Furthermore, the primary might already know the message `R` requested by the replicas. In that case, it sends the old `OR` message to the requesting replica.

Safety

Definition 3.13 (Safety)**.** *We call a system safe if the following condition holds: If a command with sequence number j and a history h_j completes, then for any command that completed earlier (with a smaller sequence number $i < j$), the history h_i is a prefix of history h_j.*

Remarks:

- In Zyzzyva a command can only complete in two ways, either in Algorithm 3.8 or in Algorithm 3.9.
- If a system is safe, complete commands cannot be reordered or dropped. So is Zyzzyva so far safe?

Lemma 3.14. *Let c_i and c_j be two different complete commands. Then c_i and c_j must have different sequence numbers.*

Proof. If a command c completes in Algorithm 3.8, $3f+1$ replicas sent a `Response`$(a,$`OR`$)_r$ to the client. If the command c completed in Algorithm 3.9, at least $2f+1$ replicas sent a `Response`$(a,$`OR`$)_r$ message to the client. Hence, a client has to receive at least $2f+1$ `Response`$(a,$`OR`$)_r$ messages.

Both c_i and c_j are complete. Therefore there must be at least $2f+1$ replicas that responded to c_i with a `Response`$(a,$`OR`$)_r$ message. But there are also at least $2f+1$ replicas that responded to c_j with a `Response`$(a,$`OR`$)_r$ message. Because there are only $3f+1$ replicas, there is at least one correct replica that sent a `Response`$(a,$`OR`$)_r$ message for both c_i and c_j. A correct replica only sends one `Response`$(a,$`OR`$)_r$ message for each sequence number, hence the two commands must have different sequence numbers. □

Lemma 3.15. *Let c_i and c_j be two complete commands with sequence numbers $i < j$. The history h_i is a prefix of h_j.*

Proof. As in the proof of Lemma 3.14, there has to be at least one correct replica that sent a `Response`$(a,$`OR`$)_r$ message for both c_i and c_j.

A correct replica r that sent a `Response`$(a,$`OR`$)_r$ message for c_i will only accept c_j if the history for c_j provided by the primary is consistent with the local history of replica r, including c_i. □

Remarks:

- A byzantine primary can cause the system to never complete any command. Either by never sending any messages or by inconsistently ordering client requests. In this case, replicas have to replace the primary.

View Changes

Definition 3.16 (View Change). *In Zyzzyva, a view change is used to replace a byzantine primary with another (hopefully correct) replica. View changes are initiated by replicas sending* `IHatePrimary`$_r$ *to all other replicas. This only happens if a replica obtains a valid proof of misbehavior from a client or after a replica fails to obtain an* `OR` *message from the primary in Algorithm 3.12.*

Remarks:

- How can we safely decide to initiate a view change, i.e. demote a byzantine primary? Note that byzantine nodes should not be able to trigger a view change!

Algorithm 3.17 Zyzzyva: View Change Agreement

```
1: All replicas continuously collect the set $H$ of IHatePrimary$_r$ messages
2: if a replica $r$ received $|H| > f$ messages or a valid ViewChange message
   then
3:     Replica $r$ broadcasts ViewChange($H^r$,$h^r$,$S_l^r$)$_r$
4:     Replica $r$ stops participating in the current view
5:     Replica $r$ switches to the next primary "$p = p + 1$"
6: end if
```

Remarks:

- The $f + 1$ `IHatePrimary`$_r$ messages in set H prove that at least one correct replica initiated a view change. This proof is broadcast to all replicas to make sure that once the first correct replica stopped acting in the current view, all other replicas will do so as well.
- S_l^r is the most recent commit certificate that the replica obtained in the ending view as described in Algorithm 3.9. S_l^r will be used to recover the correct history before the new view starts. The local histories h^r are included in the `ViewChange`(H^r,h^r,S_l^r)$_r$ message such that commands that completed after a correct client received $3f + 1$ responses from replicas can be recovered as well.
- In Zyzzyva, a byzantine primary starts acting as a normal replica after a view change. In practice, all machines eventually break and rarely fix themselves after that. Instead, one could consider to replace a byzantine primary with a fresh replica that was not in the previous view.

Algorithm 3.18 Zyzzyva: View Change Execution

```
1: The new primary $p$ collects the set $C$ of ViewChange($H^r$,$h^r$,$S_l^r$)$_r$ messages
2: if new primary $p$ collected $|C| \geq 2f + 1$ messages then
3:     New primary $p$ sends NewView($C$)$_p$ to all replicas
4: end if

5: if a replica $r$ received a NewView($C$)$_p$ message then
6:     Replica $r$ recovers new history $h_{\texttt{new}}$ as shown in Algorithm 3.20
7:     Replica $r$ broadcasts ViewConfirm($h_{\texttt{new}}$)$_r$ message to all replicas
8: end if

9: if a replica $r$ received $2f + 1$ ViewConfirm($h_{\texttt{new}}$)$_r$ messages then
10:    Replica $r$ accepts $h^r = h_{\texttt{new}}$ as the history of the new view
11:    Replica $r$ starts participating in the new view
12: end if
```

Remarks:

- Analogously to Lemma 3.15, commit certificates are ordered. For two commit certificates S_i and S_j with sequence numbers $i < j$, the history h_i certified by S_i is a prefix of the history h_j certified by S_j.

- Zyzzyva collects the most recent commit certificate and the local history of $2f + 1$ replicas. This information is distributed to all replicas, and used to recover the history for the new view h_{new}.

- If a replica does not receive the `NewView`$(C)_p$ or the `ViewConfirm`$(h_{\texttt{new}})_r$ message in time, it triggers another view change by broadcasting `IHatePrimary`$_r$ to all other replicas.

- How is the history recovered exactly? It seems that the set of histories included in C can be messy. How can we be sure that complete commands are not reordered or dropped?

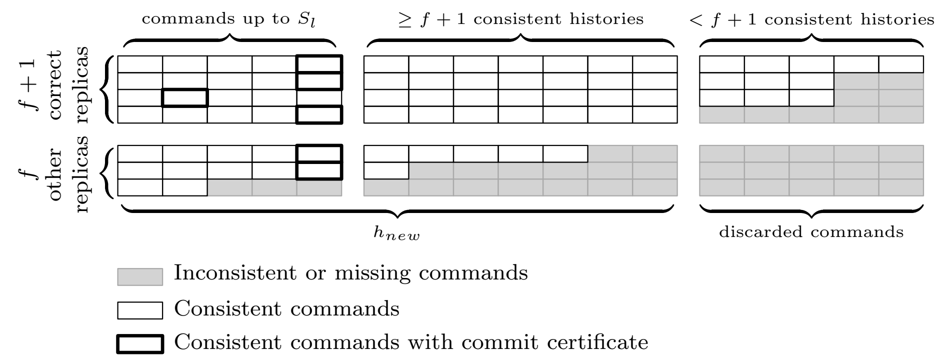
commands up to S_l
$\geq f + 1$ consistent histories
$< f + 1$ consistent histories
$f + 1$ correct replicas
f other replicas
h_{new}
discarded commands
Inconsistent or missing commands
Consistent commands
Consistent commands with commit certificate

Figure 3.19: The structure of the data reported by different replicas in C. Commands up to the last commit certificate S_l were completed in either Algorithm 3.8 or Algorithm 3.9. After the last commit certificate S_l there may be commands that completed at a correct client in Algorithm 3.8. Algorithm 3.20 shows how the new history h_{new} is recovered such that no complete commands are lost.

Algorithm 3.20 Zyzzyva: History Recovery

```
 1: $C$ = set of $2f + 1$ ViewChange($H^r$,$h^r$,$S^r$)$_r$ messages in NewView($C$)$_p$
 2: $R$ = set of replicas included in $C$
 3: $S_l$ = most recent commit certificate $S_l^r$ reported in $C$
 4: $h_{\texttt{new}}$ = history $h_l$ contained in $S_l$
 5: $k = l + 1$, next sequence number
 6: while command $c_k$ exists in $C$ do
 7:   if $c_k$ is reported by at least $f + 1$ replicas in $R$ then
 8:     Remove replicas from $R$ that do not support $c_k$
 9:     $h_{\texttt{new}} = (h_{\texttt{new}}, c_k)$
10:   end if
11:   $k = k + 1$
12: end while
13: return $h_{\texttt{new}}$
```

Remarks:

- Commands up to S_l are included into the new history h_{new}.
- If at least $f+1$ replicas share a consistent history after the last commit certificate S_l, also the commands after that are included.
- Even if $f + 1$ correct replicas consistently report a command c after the last commit certificate S_l, c may not be considered complete by a client, e.g., because one of the responses to the client was lost. Such a command is included in the new history h_{new}. When the client retries executing c, the replicas will be able to identify the same command c using the timestamp included in the client's request, and avoid duplicate execution of the command.
- Can we be sure that all commands that completed at a correct client are carried over into the new view?

Lemma 3.21. *The globally most recent commit certificate S_l is included in C.*

Proof. Any two sets of $2f+1$ replicas share at least one correct replica. Hence, at least one correct replica which acknowledged the most recent commit certificate S_l also sent a `LocalCommit`(S_l)$_r$ message that is in C. □

Lemma 3.22. *Any command and its history that completes after S_l has to be reported in C at least $f + 1$ times.*

Proof. A command c can only complete in Algorithm 3.8 after S_l. Hence, $3f+1$ replicas sent a `Response`(a,`OR`)$_r$ message for c. C includes the local histories of $2f + 1$ replicas of which at most f are byzantine. As a result, c and its history is consistently found in at least $f + 1$ local histories in C. □

Lemma 3.23. *If a command c is considered complete by a client, command c remains in its place in the history during view changes.*

Proof. We have shown in Lemma 3.21 that the most recent commit certificate is contained in C, and hence any command that terminated in Algorithm 3.9

is included in the new history after a view change. Every command that completed before the last commit certificate S_l is included in the history as a result. Commands that completed in Algorithm 3.8 after the last commit certificate are supported by at least $f+1$ correct replicas as shown in Lemma 3.22. Such commands are added to the new history as described in Algorithm 3.20. Algorithm 3.20 adds commands sequentially until the histories become inconsistent. Hence, complete commands are not lost or reordered during a view change. □

Theorem 3.24. *Zyzzyva is safe even during view changes.*

Proof. Complete commands are not reordered within a view as described in Lemma 3.15. Also, no complete command is lost or reordered during a view change as shown in Lemma 3.23. Hence, Zyzzyva is safe. □

Remarks:

- So Zyzzyva correctly handles complete commands even in the presence of failures. We also want Zyzzyva to make progress, i.e., commands issued by correct clients should complete eventually.
- If the network is broken or introduces arbitrarily large delays, commands may never complete.
- Can we be sure commands complete in periods in which delays are bounded?

Definition 3.25 (Liveness)**.** *We call a system* ***live*** *if every command eventually completes.*

Lemma 3.26. *Zyzzyva is live during periods of synchrony if the primary is correct and a command is requested by a correct client.*

Proof. The client receives a `Response`$(a,$`OR`$)_r$ message from all correct replicas. If it receives $3f+1$ messages, the command completes immediately in Algorithm 3.8. If the client receives fewer than $3f+1$ messages, it will at least receive $2f+1$, since there are at most f byzantine replicas. All correct replicas will answer the client's `Commit`$(S)_u$ message with a correct `LocalCommit`$(S)_r$ message after which the command completes in Algorithm 3.9. □

Lemma 3.27. *If, during a period of synchrony, a request does not complete in Algorithm 3.8 or Algorithm 3.9, a view change occurs.*

Proof. If a command does not complete for a sufficiently long time, the client will resend the `R` = `Request`$(c,t)_u$ message to all replicas. After that, if a replica's `ConfirmRequest(R)`$_r$ message is not answered in time by the primary, it broadcasts an `IHatePrimary`$_r$ message. If a correct replica gathers $f+1$ `IHatePrimary`$_r$ messages, the view change is initiated. If no correct replica collects more than f `IHatePrimary`$_r$ messages, at least one correct replica received a valid `OrderedRequest`$(h^p, c,$ `R`$)_p$ message from the primary which it forwards to all other replicas. In that case, the client is guaranteed to receive at least $2f+1$ `Response`$(a,$`OR`$)_r$ messages from the correct replicas and can complete the command by assembling a commit certificate. □

Remarks:

- If the newly elected primary is byzantine, the view change may never terminate. However, we can detect if the new primary does not assemble C correctly as all contained messages are signed. If the primary refuses to assemble C, replicas initiate another view change after a timeout.

Chapter Notes

Algorithm 3.2 was introduced by Dolev et al. [DFF+82] in 1982. Byzantine fault tolerant state machine replication (BFT) is a problem that gave rise to various protocols. Castro and Liskov [MC99] introduced the Practical Byzantine Fault Tolerance (PBFT) protocol in 1999, applications such as Farsite [ABC+02] followed. This triggered the development of, e.g., Q/U [AEMGG+05] and HQ [CML+06]. Zyzzyva [KAD+07] improved on performance especially in the case of no failures, while Aardvark [CWA+09] improved performance in the presence of failures. Guerraoui at al. [GKQV10] introduced a modular system which allows to more easily develop BFT protocols that match specific applications in terms of robustness or best case performance.

This chapter was written in collaboration with Pascal Bissig.

Bibliography

[ABC+02] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Oper. Syst. Rev.*, 36(SI):1–14, December 2002.

[AEMGG+05] Michael Abd-El-Malek, Gregory R Ganger, Garth R Goodson, Michael K Reiter, and Jay J Wylie. Fault-scalable byzantine fault-tolerant services. *ACM SIGOPS Operating Systems Review*, 39(5):59–74, 2005.

[CML+06] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 177–190, Berkeley, CA, USA, 2006. USENIX Association.

[CWA+09] Allen Clement, Edmund L Wong, Lorenzo Alvisi, Michael Dahlin, and Mirco Marchetti. Making byzantine fault tolerant systems tolerate byzantine faults. In *NSDI*, volume 9, pages 153–168, 2009.

[DFF+82] Danny Dolev, Michael J Fischer, Rob Fowler, Nancy A Lynch, and H Raymond Strong. An efficient algorithm for byzantine agreement without authentication. *Information and Control*, 52(3):257–274, 1982.

[GKQV10] Rachid Guerraoui, Nikola Knežević, Vivien Quéma, and Marko Vukolić. The next 700 bft protocols. In *Proceedings of the 5th European conference on Computer systems*, pages 363–376. ACM, 2010.

[KAD$^+$07] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 45–58. ACM, 2007.

[MC99] Barbara Liskov Miguel Castro. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.