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 ∈ {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 received messages contain the same value v then
8:          Broadcast propose(v, round)
9:      else
10:         Broadcast propose(⊥, 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 received 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. □

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* $f < n/3$.

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.