Chapter 16

Consensus

16.1 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' that requires fewer messages than P, contradicting the assumption that P required the minimal number of messages.

- Can Alice and Bob use Paxos?

16.2 Consensus

In Chapter 15 we studied a problem that we vaguely called agreement. We will now introduce a formally specified variant of this problem, called *consensus*.

Definition 16.1 (consensus)**.** *There are n nodes, of which at most f might crash, i.e., at least $n - f$ nodes are* ***correct****. Node i starts with an input value v_i. The nodes must decide for one of those values, satisfying the following properties:*

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

Remarks:

- We assume that every node can send messages to every other node, and that we have reliable links, i.e., a message that is sent will be received.
- There is no broadcast medium. If a node wants to send a message to multiple nodes, it needs to send multiple individual messages. If a node crashes while broadcasting, not all nodes may receive the broadcast message. Later we will call this best-effort broadcast.
- Does Paxos satisfy all three criteria? If you study Paxos carefully, you will notice that Paxos does not guarantee termination. For example, the system can be stuck forever if two clients continuously request tickets, and neither of them ever manages to acquire a majority.
- Can we solve consensus in a more benign setting where messages arrive reliably after a bounded time? We will first study consensus in such a setting.

16.3 Synchronous Consensus

Let us assume that communication proceeds in synchronous *rounds.*

Model 16.2 (synchronous)**.** *In the* ***synchronous model****, nodes operate in synchronous rounds. In each round, each node may send a message to the other nodes, receive the messages sent by the other nodes, and do some local computation.*

Remarks:

- The synchronous model is equivalent to a model with variable message delays but all messages are guaranteed to arrive within bounded time (the "round duration").
- A simple strategy is to agree on the smallest value.

Algorithm 16.3 Synchronous Consensus with $f < n$ crash failures

```
1: $v_i \in \mathbb{R}$            ◁ input
2: $min := v_i$
3: for $i = 1, \dots, f+1$ do
4:     Broadcast $min$
5:     Collect broadcast messages in set $M$
6:     $min := \min(M)$
7: end for
8: Decide on $min$
```

Theorem 16.4. *Algorithm 16.3 is a correct consensus algorithm.*

Proof. Note that the algorithm trivially satisfies the termination and validity conditions: The algorithm runs for exactly $f + 1$ rounds and the decision value must be an input value of a node because *min* is chosen from a set of broadcast input values.

It remains to show that agreement holds as well. Since the algorithm runs for $f + 1$ rounds and there are at most f failures, there must be a round without failures. In this round, all (remaining) nodes receive the local *min* value from all nodes and set *min* to the minimum of all values of these nodes. If there are more rounds, all nodes will always broadcast this minimum value and no node will ever set *min* to a different value anymore. □

Remarks:

- The algorithm is simple and works for any number $f < n$ of failures!

Definition 16.5 (synchronous runtime)**.** *For algorithms in the synchronous model, the* ***runtime*** *is the number of rounds from the start of the execution to its completion in the worst case (every legal input, every execution scenario).*

Remarks:

- Algorithm 16.3 has a synchronous runtime of $f + 1$.
- Can we do better? In the next section, we show that the answer is no.

16.4 Synchronous Lower Bound

In this section, we show that any deterministic consensus algorithm needs at least $f + 1$ rounds for any $f \leq n - 2$, even under the following validity condition using binary input.

Definition 16.6 (Validity)**.** *If all non-faulty nodes start with the same value x, the output must be x.*

Remarks:

- The proof uses executions that are indistinguishable for some nodes. Specifically, executions are transformed step by step, ensuring that in each transition from an execution E to an execution E', there is always at least one node that receives the same messages in both executions and therefore must have the same output value.
- Since this is true for every transition, the nodes must output the same value in all executions. We call executions that must have the same output at all nodes *equivalent*.
- Without loss of generality, we assume that every node sends a message to every other node in every round. For any algorithm that sends fewer messages, we can simply introduce empty "dummy messages". The same argument can be applied to algorithms where some nodes terminate before the end of round f: Such nodes execute "dummy rounds" that do nothing in the remaining rounds.

- We say that a node *fails in round* r if it fails before the messages of round r are delivered. Note that it may still send some messages in round r. We say that a node *fails completely in round* r if it fails before sending any round-r messages. Further note that there is a round $f+1$ where the nodes decide but do not send any more messages.
- We consider executions where at most one node fails in rounds $1, \ldots, f+1$. Since only f nodes are allowed to fail, we have to be careful to ensure that we only consider executions where at most f nodes fail.

Definition 16.7. *Let $\mathcal{E}^r$ denote the set of executions without any failures in rounds $r, \ldots, f+1$ and at most one failure in any round $1, \ldots, r-1$.*

Remarks:

- Given any execution $E \in \mathcal{E}^f$, we can always get an equivalent execution $E' \in \mathcal{E}^{f+1}$ by letting any node fail in round f *after* sending all messages. These executions are equivalent because the nodes receive the same set of messages in both executions and then terminate after round f.
- We use this observation to ensure that there are at most f failures: Rather than letting a node fail (completely) in round $f+1$, we let it fail in round f after sending all messages.

Lemma 16.8. *Given $E \in \mathcal{E}^{r+1}$ where v fails in round r after sending $s < n$ messages. There is an equivalent execution $E' \in \mathcal{E}^{r+1}$ where v sends $s+1$ messages in round r.*

Lemma 16.9. *Given $E \in \mathcal{E}^{r+1}$ where v fails in round r after sending $s > 0$ messages. There is an equivalent execution $E' \in \mathcal{E}^{r+1}$ where v sends $s-1$ messages in round r.*

Lemma 16.10. *Given $E \in \mathcal{E}^{r+2}$ where v fails completely in round $r+1$, there is an equivalent execution $E' \in \mathcal{E}^{r+1}$ where v fails completely in round r.*

Lemma 16.11. *Given $E \in \mathcal{E}^{r+1}$ where v fails completely in round $r \leq f$, there is an equivalent execution $E' \in \mathcal{E}^{r+2}$ where only v fails completely in round $r+1$.*

Remarks:

- We prove all lemmas together (!) using an inductive proof. First, we establish the base case for all lemmas for round f.

Lemma 16.12. *Base case: All lemmas hold for round $r = f$.*

Proof. We prove the base case for all lemmas separately.

- Lemma 16.8: In execution $E' \in \mathcal{E}^{f+1}$, v sends a message to a node w that did not receive a message from v in execution E. If w has crashed, the executions are clearly equivalent. If w has not crashed, since there are at most f failures in execution E' by definition and $n \geq f+2$, there must be at least one other node u that is alive. Since there is no additional round of messages, there is no change for u and therefore the execution is equivalent.

- Lemma 16.9: In execution $E' \in \mathcal{E}^{f+1}$, v no longer sends a message to some node w. Again, either node w has already crashed or the same argument shows that there must be at least one other node u that is not affected (because $n \geq f+2$) so the execution is equivalent.
- Lemma 16.10: Instead of failing completely in round $f+1$, let node v fail after sending all messages in round f, which is equivalent as there is no change for any node other than v. We can now apply Lemma 16.9, which holds for round $r = f$, repeatedly until all messages from node v in round r are removed. Once all messages are removed, node v fails completely in round $r = f$.
- Lemma 16.11: Since Lemma 16.8 holds for round $r = f$, we can apply the lemma repeatedly until v sends all messages before crashing. This execution is equivalent to node v crashing in round $f+1$ instead.

□

Remarks:

- Note that a node failed in round $f+1$ for the base case of Lemma 16.10 and Lemma 16.11. As mentioned before, we have to be careful not to trigger this base case by using the observation that a node may equivalently fail in round f after sending all messages.
- The assumption that $n \geq f+2$ is needed, otherwise there may be only 1 node (or no nodes) left after f crashes, in which case it cannot be argued that the remaining node must keep its value.

Lemma 16.13. *Induction step: If the lemmas hold for round r or higher, then they also hold for round $r-1$.*

Proof. We prove the induction step for all lemmas separately.

- Lemma 16.8: Consider node v that fails in round $r-1$ in execution $E \in \mathcal{E}^r$. Let w be a node that does not receive a message from v in round $r-1$. If w failed in earlier rounds, we can just add the message and get an equivalent execution. Let us assume that w is alive. This execution is equivalent to an execution where w fails in round f after sending all messages. We can apply Lemma 16.9 repeatedly to get an equivalent execution where w does not send any messages, i.e., w fails completely, in round f. Since Lemma 16.10 holds for rounds r and higher, we can apply it repeatedly until w fails completely in round r. We can now add the message from v to w, which is equivalent as w does not send any messages in round r or later rounds. Since Lemma 16.11 holds for rounds r or higher, we can apply it repeatedly until w fails completely in round f. We can then apply Lemma 16.8 repeatedly until w sends messages to all nodes before crashing. This execution is equivalent to w not failing at all. Thus, we created an equivalent execution $E' \in \mathcal{E}^r$ where v sends one more message in round $r-1$.
- Lemma 16.9: Consider node v that fails in round $r-1$ in execution $E \in \mathcal{E}^r$. Let w be a node that receives a message from v in round $r-1$. As before,

we can just remove the message if w failed in earlier rounds. Let us assume that w is alive. This execution is equivalent to an execution where w fails in round f after sending all messages. As before, we can apply 16.9 repeatedly to get an equivalent execution where w does not send any messages, i.e., w fails completely, in round f. Since Lemma 16.10 holds for rounds r and higher, we can apply it repeatedly until w fails completely in round r. We can now remove the message from v to w, which is equivalent as w does not send any messages in round r or higher. Since Lemma 16.11 holds for rounds r or higher, we can apply it repeatedly until w fails completely in round f. Once more, we can apply Lemma 16.8 repeatedly until w sends messages to all nodes before crashing. This execution is equivalent to w not failing at all. Thus, we created an equivalent execution $E' \in \mathcal{E}^r$ where v sends one message less in round $r-1$.

- Lemma 16.10: Consider node v that fails completely in round r in some execution $E \in \mathcal{E}^{r+1}$. In an equivalent execution, v fails in round $r-1$ after sending all its messages. Since Lemma 16.9 holds for round $r-1$ as shown above, we can apply it repeatedly to remove all messages sent by v and still get an equivalent execution. Once all messages have been removed, we get that v fails completely in round $r-1$.

- Lemma 16.11: Consider node v that fails completely in round $r-1$ in some execution $E \in \mathcal{E}^r$. Since Lemma 16.8 holds for round $r-1$, we can apply it repeatedly until we get an equivalent execution where v fails after sending all messages in round $r-1$. This execution is equivalent to an execution where v fails completely in round r.

□

Remarks:

- Lemma 16.10 and Lemma 16.11 may give the impression that there is only one failure that is pushed back and forth but this is not the case. Both lemmas use the other two lemmas, which themselves use Lemma 16.10 and Lemma 16.11 recursively. When looking at the intermediate executions more closely, "chains" of failures are used, with at most one failure in any round $1, \ldots, f$, i.e., there are at most f failures.

Theorem 16.14. *Any deterministic consensus algorithm in the synchronous communication model requires at least $f+1$ rounds if there are $n \geq f+2$ nodes.*

Proof. We start with an execution where every node has input 0. According to Definition 16.6, all nodes must output 0. The execution is equivalent to the execution that is identical except for some node v failing in round f after sending all messages. We can first apply Lemma 16.9 repeatedly until we get an execution where v does not send any messages anymore and then apply Lemma 16.10 repeatedly until we get an equivalent execution where v fails completely in round 0. We can then change its input to 1, which is again an equivalent execution because v fails right at the start of the execution. Then, we apply Lemma 16.11 repeatedly until v fails completely in round f. We can then apply Lemma 16.8 repeatedly to get an execution where v fails after sending all its messages. This execution is equivalent to an execution that is identical except for the fact that v does not fail at all.

We can then repeat this process with the other $n - 1$ nodes, resulting in an equivalent execution where no node fails and all inputs are 1. However, since the executions are all equivalent, the output of the nodes is still 0. Thus, the validity condition is violated as it requires that the nodes output 1 in this scenario. □

16.5 Impossibility of Consensus

Let's consider again the model where message delays are unbounded. Note that Paxos works in this model but does not terminate. One may hope to fix Paxos somehow, to guarantee termination. However, this is impossible. In fact, the consensus problem of Definition 16.1 cannot be solved by any algorithm in the *asynchronous model*.

Model 16.15 (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 15.6).

Definition 16.16 (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 (i.e., 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 16.17 (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 16.18 (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 16.19 (bivalent)**.** *A configuration C is called* ***bivalent*** *if the nodes might decide for* 0 *or* 1*.*

Remarks:

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

Lemma 16.20. *There is at least one selection of input values V such that the corresponding initial configuration C_0 is bivalent, if $f \geq 1$.*

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

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

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

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

□

Definition 16.21 (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 16.22 (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 16.23. *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 $C_{\tau_1\tau_2} = C_{\tau_2\tau_1}$. $\square$

Definition 16.24 (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 16.25. *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 16.20). Assuming that this configuration is not critical, there must be at least one following configuration that is bivalent; hence, the system may enter this configuration. But if this configuration is not critical either, 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 that 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 16.26. *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 16.23 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 $\tau = (u, m) \in T$ that leads to a 0-valent configuration. As shown before, all transitions in T that 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 16.27. *There is no deterministic algorithm that 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 16.20 we know that there must be at least one bivalent initial configuration C. Using Lemma 16.25 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 16.26). □

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.

16.6 Randomized Consensus

Algorithm 16.28 Randomized Consensus (assuming $f < n/2$)

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

Remarks:

- The idea of Algorithm 16.28 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 by chance get the same outcome.

Lemma 16.29. *As long as no node decides and terminates, Algorithm 16.28 does not get stuck, independent of which nodes crash.*

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

Lemma 16.30. *Algorithm 16.28 satisfies the validity requirement.*

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

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

Lemma 16.31. *Algorithm 16.28 satisfies the agreement requirement.*

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

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

In Lemma 16.29 we only showed that nodes do not get stuck as long as no node decides, thus we need to be careful that no node gets stuck if u terminates.

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

Lemma 16.32. *Algorithm 16.28 satisfies the termination requirement, i.e., all nodes terminate in expected time $O(2^n)$.*

Proof. We know from the proof of Lemma 16.31 that once a node hears a majority of proposals for a value, all nodes will terminate at most one round later. Hence, we only need to show that a node receives a majority of proposals for the same value within expected time $O(2^n)$.

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

Theorem 16.33. *Algorithm 16.28 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 16.34. *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 16.28 solves consensus with optimal fault-tolerance but it is awfully slow. The problem is rooted in the individual coin tossing: If all nodes toss the same coin, they could terminate in a constant number of rounds.
- Can this problem be fixed by simply always choosing 1 at Line 19?!
- This cannot work: Such a change makes the algorithm deterministic, and therefore it cannot achieve consensus (Theorem 16.27). 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*, which we will discuss in Chapter 19. In short, 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 16.28, we replace Line 19 with a function call to the shared coin algorithm.

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* [LSP82]. The impossibility proof was established in 1975 by Akkoyunlu et al. [AEH75].

Dolev and Strong proved that any deterministic consensus algorithms requires at least $f + 1$ rounds in the synchronous model [DS83] based on a more complicated proof from Fischer and Lynch [FL82].

The proof that there is no deterministic algorithm that always solves consensus in the asynchronous model even if there is at most one crash failure 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].

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

This chapter was written in collaboration with David Stolz and Thomas Locher.

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.

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

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

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

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

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

[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.

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