Chapter 8

Consensus

This chapter is the first to deal with fault tolerance, one of the most fundamental aspects of distributed computing. Indeed, in contrast to a system with a single processor, having a distributed system may permit getting away with failures and malfunctions of parts of the system. This line of research was motivated by the basic question whether, e.g., putting two (or three?) computers into the cockpit of a plane will make the plane more reliable. Clearly fault-tolerance often comes at a price, as having more than one decision-maker often complicates decision-making.

8.1 Impossibility of Consensus

Imagine two cautious generals who want to attack a common enemy.[1] Their only means of communication are messengers. Unfortunately, the route of these messengers leads through hostile enemy territory, so there is a chance that a messenger does not make it. Only if both generals attack at the very same time the enemy can be defeated. Can we devise a protocol such that the two generals can agree on an attack time? Clearly general A can send a message to general B asking to e.g. "attack at 6am". However, general A cannot be sure that this message will make it, so she asks for a confirmation. The problem is that general B getting the message cannot be sure that her confirmation will reach general A. If the confirmation message indeed is destroyed, general A cannot distinguish this case from the case where general B did not even get the attack information. So, to be save, general B herself will ask for a confirmation of her confirmation. Taking again the position of general A we can similarly derive that she cannot be sure unless she also gets a confirmation of the confirmation of the confirmation...

To make things worse, also different approaches do not seem to work. In fact it can be shown that this two generals problem cannot be solved, in other words, there is no finite protocol which lets the two generals find consensus! To show this, we need to be a bit more formal:

[1] If you don't fancy the martial tone of this classic example, feel free to think about something else, for instance two friends trying to make plans for dinner over instant messaging software, or two lecturers sharing the teaching load of a course trying to figure out who is in charge of the next lecture.

Definition 8.1 (Consensus)**.** *Consider a distributed system with n nodes. Each node i has an input x_i. A solution of the* consensus *problem must guarantee the following:*

- *Termination: Every non-faulty node eventually decides.*
- *Agreement: All non-faulty nodes decide on the same value.*
- *Validity: The decided value must be the input of at least one node.*

Remarks:

- The validity condition infers that if all nodes have the same input x, then the nodes need to decide on x. Please note that consensus is not democratic, it may well be that the nodes decide on an input value promoted by a small minority.
- Whether consensus is possible depends on many parameters of the distributed system, in particular whether the system is synchronous or asynchronous, or what "faulty" means. In the following we study some simple variants to get a feeling for the problem.
- Consensus is a powerful primitive. With established consensus almost everything can be computed in a distributed system, e.g. a leader.

Given a distributed asynchronous message passing system with $n \geq 2$ nodes. All nodes can communicate directly with all other nodes, simply by sending a message. In other words, the communication graph is the complete graph. Can the consensus problem be solved? Yes!

Algorithm 28 Trivial Consensus

```
Each node has an input
We have a leader, e.g. the node with the highest ID
if node $v$ is the leader then
    the leader shall simply decide on its own input
else
    send message to the leader asking for its input
    wait for answer message by leader, and decide on that
end if
```

Remarks:

- This algorithm is quite simple, and at first sight seems to work perfectly, as all three consensus conditions of Definition 8.1 are fulfilled.
- However, the algorithm is not fault-tolerant at all. If the leader crashes before being able to answer all requests, there are nodes which will never terminate, and hence violate the termination condition. Is there a deterministic protocol that can achieve consensus in an asynchronous system, even in the presence of failures? Let's first try something slightly different.

Definition 8.2 (Reliable Broadcast)**.** *Consider an asynchronous distributed system with n nodes that may crash. We want node v to send a* reliable broadcast *to the $n-1$ other nodes. Reliable means that either nobody receives the message, or everybody receives the message.*

Remarks:

- This seems to be quite similar to consensus, right?
- The main problem is that the sender may crash while sending the message to the $n-1$ other nodes such that some of them get the message, and the others not. We need a technique that deals with this case:

Algorithm 29 Reliable Broadcast

```
if node $v$ is the source of message $m$ then
    send message $m$ to each of the $n-1$ other nodes
    upon receiving $m$ from any other node: broadcast succeeded!
else
    upon receiving message $m$ for the first time:
    send message $m$ to each of the $n-1$ other nodes
end if
```

Theorem 8.3. *Algorithm 29 solves reliable broadcast as in Definition 8.2.*

Proof. First we should note that we do not care about nodes that crash during the execution: whether or not they receive the message is irrelevant since they crashed anyway. If a single non-faulty node u received the message (no matter how, it may be that it received it through a path of crashed nodes) all non-faulty nodes will receive the message through u. If no non-faulty node receives the message, we are fine as well! □

Remarks:

- While it is clear that we could also solve reliable broadcast by means of a consensus protocol (first send message, then agree on having received it), the opposite seems more tricky!
- No wonder, it cannot be done!! Traditionally this impossibility result is presented in a read/write shared memory model. In our lecture, we stick to the asynchronous message passing model, however we borrow the terminology and proof structure from the classic shared memory proof:

Definition 8.4 (Univalent, Bivalent)**.** *A distributed system is called x-*valent *if the outcome of a computation will be x. An x-valent system is also called univalent. If, depending on the execution, still more than one possible outcome is feasible, the system is called* multivalent. *If exactly two outcomes are still possible, the system is called* bivalent.

Theorem 8.5. *In an asynchronous message passing system with $n > 1$ nodes, and node crash failures (but no message failures!) consensus as in Definition 8.1 cannot be achieved by a deterministic algorithm.*

Proof. Let us simplify the proof by setting $n = 2$. We have nodes u and v, with input values x_u and x_v. Further let the input values be binary, either 0 or 1.

First we have to make sure that there are input values such that initially the system is bivalent. If $x_u = 0$ and $x_v = 0$ the system is 0-valent, because of the validity condition (Definition 8.1). Even in the case where node v immediately crashes the system remains 0-valent. Similarly if both input values are 1 and node u immediately crashes the system is 1-valent. If $x_u = 0$ and $x_v = 1$ and v immediately crashes, node u cannot distinguish from both having input 0, equivalently if u immediately crashes, node v cannot distinguish from both having 1, hence the system is bivalent!

In order to solve consensus an algorithm needs to terminate. All non-faulty nodes need to decide on the same value x (agreement condition of Definition 8.1), in other words, at some instant this value x must be known to the system as a whole, meaning that no matter what the execution is, the system will be x-valent. In other words, the system needs to change from bivalent to univalent. We may ask ourselves what can cause this change in a deterministic asynchronous message passing algorithm? We need an element of non-determinism; if everything happens deterministically the system would have been x-valent even after initialization which we proved to be impossible already.

The only nondeterministic element in our model is the delivery of messages; if more than one message is in transit the scheduler may select an arbitrary one to be received. In other words, we hope for a *critical* bivalent state with more than one message in transit, depending which message arrives next the system is going to switch from bivalent to univalent. More concretely, if message m is being received next the system is going x-valent, if message m' (with $m' \neq m$) is going to be received next the system is going x'-valent, with $x' \neq x$. We have two cases:

- If both m and m' are being received by different receiver nodes u and v we are in trouble, since they might be scheduled shortly after each other. Since no node can distinguish the ordering of the two messages, we cannot have had a critical state. The system remains bivalent.

- If both messages are being received by the same node u we are equally in trouble: node u may crash, and no other node in the system can distinguish x-valent from x'-valent.

Hence, the system will remain bivalent forever, and consensus is impossible. □

Remarks:

- The proof presented is a variant of a proof by Michael Fischer, Nancy Lynch and Michael Paterson, a classic result in distributed computing. The proof was motivated by the problem of committing transactions in distributed database systems, but is sufficiently general that it directly implies the impossibility of a number of related problems, including consensus. The proof also is pretty robust with regard to different communication models.

- The FLP (Fischer, Lynch, Paterson) paper won the 2001 PODC Influential Paper Award, which later was renamed Dijkstra Prize.

- One might argue that FLP destroys all the fun in distributed computing, as it makes so many things impossible! For instance, it seems impossible to have a distributed database where the nodes can reach consensus whether to commit a transaction or not.
- So are two-phase-commit (2PC), three-phase-commit (3PC) et al. wrong?! No, not really, but sometimes they just do not commit!
- What about turning some other knobs of the model? Can we have consensus in synchronous systems? Yes, even if all but one node fails!
- Can we have consensus in synchronous systems even if some nodes are mischievous, and behave much worse than simply crashing, and send for example contradicting information to different nodes? (This is generally known as *Byzantine* behavior.) Yes, this is also possible, as long as the Byzantine nodes are strictly less than a third of all the nodes. This was shown by Marshall Pease, Robert Shostak, and Leslie Lamport in 1980. This paper won the 2005 Dijkstra Prize, and is one of the cornerstones not only in distributed computing but also information security. Indeed this work was motivated by the "fault-tolerance in planes" example. Pease, Shostak, and Lamport noticed that the computers they were given to implement a fault-tolerant fighter plane at times behaved strangely. Before crashing, these computers would start behaving quite randomly, sending out weird messages. At some point Pease et al. decided that a malicious behavior model would be the most appropriate to be on the save side. Being able to allow strictly less than a third Byzantine nodes is quite counterintuitive; even today many systems are built with three copies. In light of the result of Pease et al. this is a serious mistake! If you want to be tolerant against a single Byzantine fault, you need four copies, not three!
- Finally, FLP only prohibits deterministic algorithms! So can we solve consensus if we use randomization? The answer again is yes! We will study this in the remainder of this chapter.

8.2 Randomized Consensus

Can we solve consensus if we allow randomization? Yes. In this section we again study crash failures only (even though our algorithms can be extended to work with Byzantine failures as well). The general idea is that nodes try to push their input value; if other nodes do not follow they will try to push one of the suggested values randomly. To simplify arguments we assume that at most f nodes will fail (crash) with $n > 9f$, and that we only solve binary consensus, that is, the input values are 0 and 1. The full algorithm is in Algorithm 30.

Theorem 8.6. *Algorithm 30 solves consensus as in Definition 8.1.*

Proof. If all nodes have the same input value x, then all nodes will propose the same value x, all will bid for the same value x, and decide on the same value x in the first round already. We have consensus!

If the nodes have different (binary) input values the validity condition becomes trivial as any result is fine. What about agreement? Let u be one of

Algorithm 30 Randomized Consensus

```
1: node $u$ starts with input bit $x_u \in \{0,1\}$, round:=1.
2: repeat
3:   broadcast PROPOSAL($x_u$,round)
4:   wait for $n-f$ PROPOSAL messages
5:   if at least $n-2f$ messages have value $v$ then
6:     $x_u := v$
7:   else
8:     $x_u :=$ undecided
9:   end if
10:  broadcast BID($x_u$,round)
11:  wait for $n-f$ BID messages
12:  if at least $n-2f$ messages have value $v$ then
13:    decide on $v$
14:  else if at least $n-4f$ messages have value $v$ then
15:    $x_u := v$
16:  else
17:    choose $x_u$ randomly, with $Pr[x_u = 0] = Pr[x_u = 1] = 1/2$
18:  end if
19:  round := round + 1
20: until decided
```

the first nodes to decide on value v (in line 13). It may happen that due to asynchronicity another node u' waited for a different subset of BID messages, in fact up to f BID messages may be different. However, since node u had at least $n-2f$ BID messages with value v, node u' has at least $n-3f$ BID messages with v. Hence everybody will propose v in the next round, and then decide on v.

So we only need to worry about termination! We already have seen that as soon one node terminates (in line 13) everybody terminates in the next round. So what are the chances that some node u terminates in line 13? Well, if push comes to shove we can still hope that all nodes randomly propose the same value (in line 17). This is going to happen with probability at least 2^{n-1}. If so, all will send the same PROPOSAL, and the algorithm terminates. So the expected running time is $O(2^n)$. □

Remarks:

- The presentation of Algorithm 30 follows the typical presentation in text books; it can be simplified (see exercises).

- What about an algorithm that allows Byzantine faults? Good news! The presented algorithm already does that! That's why we have "$n-4f$" in the algorithm and "$n-3f$" in the proof.

- Unfortunately Algorithm 30 is still impractical as termination is awfully slow. In expectation about the same number of nodes choose 1 or 0 in line 17. Termination would be much more efficient if all nodes chose the same random value in line 17! So why not simply replacing line 17 with "choose $x_u := 1$"?!? The problem is that a majority of nodes may see a majority of 0 bids, hence proposing 0 in the next round. Without randomization it is impossible to get out of this equilibrium. (Moreover, this approach is deterministic, contradicting Theorem 8.5.)

- The idea is to replace line 17 with a subroutine where all nodes compute a so-called *shared* (or common, or global) coin. A shared coin is a random variable that is 0 with constant probability and 1 with constant probability. Sounds like magic, but it isn't! We assume $n > 3f$:

Algorithm 31 Shared Coin (code for node u)

```
1: set local coin $x_u := 0$ with probability $1/n$, else $x_u := 1$
2: use reliable broadcast to tell everybody about your local coin $x_u$
3: memorize all coins you get by others in the set $c_u$
4: wait for exactly $n - f$ coins
5: copy these coins into your local set $s_u$ (but keep extending your set of coins $c_u$)
6: use reliable broadcast to tell everybody about your set $s_u$
7: wait for exactly $n - f$ sets $s_v$ which satisfy $s_v \subseteq c_u$
8: if seen at least a single coin 0 then
9:     return 0
10: else
11:     return 1
12: end if
```

Theorem 8.7. *With $n > 3f$ Algorithm 31 implements a shared coin.*

Proof. For simplicity we consider crash failures only. Since only f nodes may crash, each node sees at least $n - f$ coins and sets in lines 4 resp. 7. Thanks to the reliable broadcast protocol each node eventually sees all the coins in the other sets. In other words, the algorithm terminates in $O(1)$ time.

The general idea is that a third of the coins are being seen by everybody. If there is a 0 among these coins, everybody will see that 0. If not chances are high that there is no 0 at all! Here are the details:

Let u be the first node to terminate (satisfy line 7). For u we draw a matrix of all the seen sets s_v (columns) and all coins c_u seen by node u (rows). Here is an example with $n = 7, f = 2, n - f = 5$:

	s_1	s_3	s_5	s_6	s_7
c_1	X	X	X	X	X
c_2			X	X	X
c_3	X	X	X	X	X
c_5	X	X	X		X
c_6	X	X	X	X	
c_7	X	X		X	X

Note that there are exactly $(n - f)^2$ X's in this matrix as node u has seen exactly $n - f$ sets (line 7) each having exactly $n - f$ coins (lines 4 to 6). We need two little helper lemmas:

Lemma 8.8. *There are at least $f + 1$ rows that have at least $f + 1$ X's*

Proof. Assume (for the sake of contradiction) that this is not the case. Then at most f rows have all $n - f$ X's, and all other rows (at most $n - f$) have at most f X's. In other words, the number of total X's is bounded by

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

Using $n > 3f$ we get $n - f > 2f$, and hence $|X| \leq 2f(n - f) < (n - f)^2$. This is a contradiction to having exactly $(n - f)^2$ X's! □

Lemma 8.9. *Let W be the set of local coins for which the corresponding matrix row has more than f X's. All local coins in the set W are seen by all nodes that terminate.*

Proof. Let $w \in W$ be such a local coin. By definition of W we know that w is in at least $f + 1$ seen sets. Since each node must see at least $n - f$ seen sets before terminating, each node has seen at least one of these sets, and hence w is seen by everybody terminating. □

Continuing the proof of Theorem 8.2: With probability $(1 - 1/n)^n \approx 1/e \approx .37$ all nodes chose their local coin equal to 1, and 1 is decided. With probability $1 - (1 - 1/n)^{|W|}$ there is at least one 0 in W. With Lemma 8.8 we know that $|W| \approx n/3$, hence the probability is about $1 - (1 - 1/n)^{n/3} \approx 1 - (1/e)^{1/3} \approx .28$. With Lemma 8.9 this 0 is seen by all, and hence everybody will decide 0. So indeed we have a shared coin. □

Theorem 8.10. *Plugging Algorithm 31 into Algorithm 30 we get a randomized consensus algorithm which finishes in a constant expected number of rounds.*

Remarks:

- If some nodes go into line 15 of Algorithm 30 the others still have a constant probability to guess the same shared coin.
- For crash failures there exists an improved constant expected time algorithm which tolerates f failures with $2f < n$.
- For Byzantine failures there exists a constant expected time algorithm which tolerates f failures with $3f < n$.
- Similar algorithms have been proposed for the shared memory model; we will study this model (for different problems, however) later.