Chapter 10

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.

10.1 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 safe, 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 10.1 (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 41 Trivial Consensus

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

Remarks:

- This algorithm is quite simple, and at first sight seems to work perfectly, as all three consensus conditions of Definition 10.1 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 10.2 (Reliable Broadcast)**.** *Consider an asynchronous distributed system with n nodes that may crash. Any two nodes can exchange messages, i.e., the communication graph is complete. 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 42 Reliable Broadcast

```
1: if node v is the source of message m then
2:    send message m to each of the n − 1 other nodes
3:    upon receiving m from any other node: broadcast succeeded!
4: else
5:    upon receiving message m for the first time:
6:    send message m to each of the n − 1 other nodes
7: end if
```

Theorem 10.3. *Algorithm 42 solves reliable broadcast as in Definition 10.2.*

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!! We will next prove this classic impossibility result. For the presentation of the result, we use a read/write shared memory model instead of a message passing model. Not only was the proof originally conceived in the shared memory model, it is also cleaner. Before proving the impossibility of consensus, we therfore first briefly discuss the shared memory model for distributed computations.

So far, the focus of the course was on loosely-coupled distributed systems such as the Internet, where nodes asynchronously communicate by exchanging messages. The "opposite" model is a tightly-coupled parallel computer where nodes access a common memory totally synchronously—in distributed computing such a system is called a Parallel Random Access Machine (PRAM).

A third major model is somehow between these two extremes, the *shared memory* model. In a shared memory system, asynchronous processes (or processors) communicate via a common memory area of shared variables or registers:

Definition 10.4 (Read/Write Shared Memory)**.** *A read/write shared memory system is a system that consists of asynchronous processes that access a common (shared) memory. A process can atomically access a register in the shared memory and either read the value in the register or write a value to the register. An atomic modification appears to the rest of the system instantaneously. Apart from this shared memory, processes can also have some local (private) memory.*

Remarks:

- Asynchronous processes means that the processes do not have a notion of time and they are scheduled by an adversary in an arbitrary order (i.e., at each point in time, the adversary can activate an arbitrary active process).
- Various alternative shared memory systems exist. A main difference is how they allow processes to access the shared memory. All systems can atomically read or write a shared register R. Most systems do allow for advanced *atomic* read-modify-write (RMW) operations, for example:
  - test-and-set(R): $t := R$; $R := 1$; return t
  - fetch-and-add(R, x): $t := R$; $R := R + x$; return t
  - compare-and-swap(R, x, y): if $R = x$ then $R := y$; return **true**; else return **false**; endif;
  - load-link(R)/store-conditional(R, x): Load-link returns the current value of the specified register R. A subsequent store-conditional to the same register will store a new value x (and return **true**) only if no updates have occurred to that register since the load-link. If any updates have occurred, the store-conditional is guaranteed to fail (and return **false**), even if the value read by the load-link has since been restored.
- The main advantage of adding such more powerful RMW operations is that consensus becomes solvable. In fact, the power of RMW operations can be measured with the so-called *consensus-number*: The consensus-number k of a RMW operation defines whether one can solve consensus for k processes. Test-and-set for instance has consensus-number 2 (one can solve consensus with 2 processes, but not 3), whereas the consensus-number of compare-and-swap is infinite. It can be shown that the power of a shared memory system is determined by the consensus-number ("universality of consensus".) This insight has a remarkable theoretical and practical impact. In practice for instance, after this was known, hardware designers stopped developing shared memory systems supporting weak RMW operations.
- Many of the results derived in the message passing model have an equivalent in the shared memory model. Consensus for instance is traditionally studied in the shared memory model.

- Whereas programming a message passing system is rather tricky (in particular if fault-tolerance has to be integrated), programming a shared memory system is generally considered easier, as programmers are given access to global variables directly and do not need to worry about exchanging messages correctly. Because of this, even distributed systems which physically communicate by exchanging messages can often be programmed through a shared memory middleware, making the programmer's life easier.

- We will most likely find the general spirit of shared memory systems in upcoming multi-core architectures. As for programming style, the multi-core community seems to favor an accelerated version of shared memory, *transactional memory.*

- From a message passing perspective, the shared memory model is like a bipartite graph: On one side you have the processes (the nodes) which pretty much behave like nodes in the message passing model (asynchronous, maybe failures). On the other side you have the shared registers, which just work perfectly (no failures, no delay).

Definition 10.5 (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 10.6. *In an asynchronous shared memory system with* $n > 1$ *nodes, and node crash failures (but no memory failures!) consensus as in Definition 10.1 cannot be achieved by a deterministic algorithm.*

Proof. Let us simplify the proof by setting $n = 2$. We have processes 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 10.1). Even in the case where process v immediately crashes the system remains 0-valent. Similarly if both input values are 1 and process u immediately crashes the system is 1-valent. If $x_u = 0$ and $x_v = 1$ and v immediately crashes, process u cannot distinguish from both having input 0, equivalently if u immediately crashes, process 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 processes need to decide on the same value x (agreement condition of Definition 10.1), 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 shared memory 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 elements in our model are the asynchrony of accessing the memory and crashing processes. Initially and after every memory

access, each process decides what to do next: Read or write a memory cell or terminate with a decision. We take control of the scheduling, either choosing which request is served next or making a process crash. Now we hope for a *critical* bivalent state with more than one memory request, and depending which memory request is served next the system is going to switch from bivalent to univalent. More concretely, if process u is being served next the system is going x-valent, if process v (with $v \neq u$) is served next the system is going y-valent (with $y \neq x$). We have several cases:

- If the operations of processes u and v target different memory cells, processes cannot distinguish which memory request was executed first. Hence the local states of the processes are identical after serving both operations and the state cannot be critical.
- The same argument holds if both processes want to read the same register. Nobody can distinguish which read was first, and the state cannot be critical.
- If process u reads memory cell c, and process v writes memory cell c, the scheduler first executes u's read. Now process v cannot distinguish whether that read of u did or did not happen before its write. If it did happen, v should decide on x, if it did not happen, v should decide y. But since v does not know which one is true, it needs to be informed about it by u. We prevent this by making u crash. Thus the state can only be univalent if v never decides, violating the termination condition!
- Also if both processes write the same memory cell we have the same issue, since the second writer will immediately overwrite the first writer, and hence the second writer cannot know whether the first write happened at all. Again, the state cannot be critical.

Hence, if we are unlucky (and in a worst case, we are!) there is no critical state. In other words, 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 a message passing system? No. 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 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. Their work 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 safe 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 machine, 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.

10.2 Randomized Consensus

Can we solve consensus if we allow randomization? Yes. The following algorithm solves Consensus even in face of Byzantine errors, i.e., malicious behavior of some of the nodes. 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 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. The full algorithm is in Algorithm 43.

Theorem 10.7. *Algorithm 43 solves consensus as in Definition 10.1 even if up to $f < n/9$ nodes exhibit Byzantine failures.*

Proof. First note that it is not a problem to wait for $n - f$ BID messages in line 4 since at most f nodes are corrupt. If all nodes have the same input value x, then all (except the f Byzantine nodes) will bid for the same value x. Thus, every node receives at least $n - 2f$ BID messages containing x, deciding on 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 43 Randomized Consensus

```
1: node u starts with input bit x_u ∈ {0,1}, round:=1.
2: broadcast BID(x_u, round)
3: repeat
4:    wait for n − f BID messages of current round
5:    if at least n − 2f messages have value x then
6:       x_u := x; decide on x
7:    else if at least n − 4f messages have value x then
8:       x_u := x
9:    else
10:      choose x_u randomly, with Pr[x_u = 0] = Pr[x_u = 1] = 1/2
11:   end if
12:   round := round + 1
13:   broadcast BID(x_u, round)
14: until decided
```

the first nodes to decide on value x (in line 6). It may happen that due to asynchronicity another node v received messages from a different subset of the nodes, however, at most f senders may be different. Taking into account that Byzantine nodes may lie, i.e., send different BIDs to different nodes, f additional BID messages received by v may differ from those received by u. Since node u had at least $n - 2f$ BID messages with value x, node v has at least $n - 4f$ BID messages with x. Hence every correct node will bid for x in the next round, and then decide on x.

So we only need to worry about termination! We already have seen that as soon as one correct node terminates (in line 6) everybody terminates in the next round. So what are the chances that some node u terminates in line 6? Well, if push comes to shove we can still hope that all correct nodes randomly propose the same value (in line 10). Maybe there are some nodes not choosing at random (i.e., entering line 8), but they unanimously propose either 0 or 1: For the sake of contradiction, assume that both 0 and 1 are proposed in line 8. This means that both 0 and 1 had been proposed by at least $n - 5f$ *correct* nodes. In other words, we have a total of $2(n - 5f) + f = n + (n - 9f) > n$ nodes. Contradiction!

Thus, at worst all $n - f$ correct nodes need to randomly choose the same bit, which happens with probability $2^{-(n-f)}$. If so, all will send the same BID, and the algorithm terminates. So the expected running time is smaller than 2^n. □

Remarks:

- The presentation of Algorithm 43 is a simplification of the typical presentation in text books.
- What about an algorithm that allows for crashes only, but can manage more failures? Good news! Slightly changing the presented algorithm will do that for $f < n/4$! See exercises.
- Unfortunately Algorithm 43 is still impractical as termination is awfully slow. In expectation about the same number of nodes choose 1 or 0 in line 10. Termination would be much more efficient if all nodes

chose the same random value in line 10! So why not simply replacing line 10 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 10.6.)

- The idea is to replace line 10 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 at most $f < n/3$ nodes may crash:

Algorithm 44 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 from 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 learning coins)
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 ⊆ c_u)
8: if seen at least a single coin 0 then
9:    return 0
10: else
11:   return 1
12: end if
```

Theorem 10.8. *If $f < n/3$ nodes crash, Algorithm 44 implements a shared coin.*

Proof. Since only f nodes may crash, each node sees at least $n - f$ coins and sets in lines 4 and 7, respectively. Thanks to the reliable broadcast protocol each node eventually sees all the coins in the other sets. In other words, the algorithm terminates in $\mathcal{O}(1)$ 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 10.9. *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 10.10. *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 10.8: 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 10.9 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 10.10 this 0 is seen by all, and hence everybody will decide 0. So indeed we have a shared coin. □

Theorem 10.11. *Plugging Algorithm 44 into Algorithm 43 we get a randomized consensus algorithm which finishes in a constant expected number of rounds.*

Remarks:

- If some nodes go into line 8 of Algorithm 43 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.

Chapter Notes

See [Lam82, FLP85, PLS83, Sim88].

Bibliography

[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.

[Lam82] L. Lamport. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4:382–401, 1982.

[PLS83] Robert L. Probert, Nancy A. Lynch, and Nicola Santoro, editors. *Proceedings of the Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Montreal, Quebec, Canada, August 17-19, 1983*. ACM, 1983.

[Sim88] Janos Simon, editor. *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*. ACM, 1988.