Chapter 11

Impossibility of asynchronous agreement

The Fischer-Lynch-Paterson (FLP) result [FLP85] says that you can't do agreement in an asynchronous message-passing system if even one crash failure is allowed, unless you augment the basic model in some way, e.g. by adding randomization or failure detectors. After its initial publication, it was quickly generalized to other models including asynchronous shared memory [LAA87], and indeed the presentation of the result in [Lyn96, §12.2] is given for shared-memory first, with the original result appearing in [Lyn96, §17.2.3] as a corollary of the ability of message passing to simulate shared memory. In these notes, I'll present the original result; the dependence on the model is surprisingly limited, and so most of the proof is the same for both shared memory (even strong versions of shared memory that support e.g. atomic snapshots[1]) and message passing.

Section 5.3 of [AW04] gives a very different version of the proof, where it is shown first for two processes in shared memory, then generalized to n processes in shared memory by adapting the classic Borowsky-Gafni simulation [BG93] to show that two processes with one failure can simulate n processes with one failure. This is worth looking at (it's an excellent example of the power of simulation arguments, and BG simulation is useful in many other contexts) but we will stick with the original argument, which is simpler. We will look at this again when we consider BG simulation in Chapter 27.

[1] Chapter 19.

11.1 Agreement

Usual rules: **agreement** (all non-faulty processes decide the same value), **termination** (all non-faulty processes eventually decide some value), **validity** (for each possible decision value, there an execution in which that value is chosen). Validity can be tinkered with without affecting the proof much.

To keep things simple, we assume the only two decision values are 0 and 1.

11.2 Failures

A failure is an internal action after which all send operations are disabled. The adversary is allowed one failure per execution. Effectively, this means that any group of $n-1$ processes must eventually decide without waiting for the n-th, because it might have failed.

11.3 Steps

The FLP paper uses a notion of *steps* that is slightly different from the send and receive actions of the asynchronous message-passing model we've been using. Essentially a step consists of receiving zero or more messages followed by doing a finite number of sends. To fit it into the model we've been using, we'll define a step as either a pair (p, m), where p receives message m and performs zero or more sends in response, or $(p, \perp)$, where p receives nothing and performs zero or more sends. We assume that the processes are deterministic, so the messages sent (if any) are determined by p's previous state and the message received. Note that these steps do not correspond precisely to delivery and send events or even pairs of delivery and send events, because what message gets sent in response to a particular delivery may change as the result of delivering some other message; but this won't affect the proof.

The fairness condition essentially says that if (p, m) or $(p, \perp)$ is continuously enabled it eventually happens. Since messages are not lost, once (p, m) is enabled in some configuration C, it is enabled in all successor configurations until it occurs; similarly $(p, \perp)$ is always enabled. So to ensure fairness, we have to ensure that any non-faulty process eventually performs any enabled step.

Comment on notation: I like writing the new configuration reached by applying a step e to C like this: Ce. The FLP paper uses $e(C)$.

11.4 Bivalence and univalence

The core of the FLP argument is a strategy allowing the adversary (who controls scheduling) to steer the execution away from any configuration in which the processes reach agreement. The guidepost for this strategy is the notion of **bivalence**, where a configuration C is **bivalent** if there exist traces T_0 and T_1 starting from C that lead to configurations CT_0 and CT_1 where all processes decide 0 and 1 respectively. A configuration that is not bivalent is **univalent**, or more specifically **0-valent** or **1-valent** depending on whether all executions starting in the configuration produce 0 or 1 as the decision value. (Note that bivalence or univalence are the only possibilities because of termination.) The important fact we will use about univalent configurations is that any successor to an x-valent configuration is also x-valent.

It's clear that any configuration where some process has decided is not bivalent, so if the adversary can keep the protocol in a bivalent configuration forever, it can prevent the processes from ever deciding. The adversary's strategy is to start in an initial bivalent configuration C_0 (which we must prove exists) and then choose only bivalent successor configurations (which we must prove is possible). A complication is that if the adversary is only allowed one failure, it must eventually allow any message in transit to a non-faulty process to be received and any non-faulty process to send its outgoing messages, so we have to show that the policy of avoiding univalent configurations doesn't cause problems here.

11.5 Existence of an initial bivalent configuration

We can specify an initial configuration by specifying the inputs to all processes. If one of these initial configurations is bivalent, we are done. Otherwise, let C and C' be two initial configurations that differ only in the input of one process p; by assumption, both C and C' are univalent. Consider two executions starting with C and C' in which process p is faulty; we can arrange for these executions to be indistinguishable to all the other processes, so both decide the same value x. It follows that both C and C' are x-valent. But since any two initial configurations can be connected by some chain of such indistinguishable configurations, we have that all initial configurations are x-valent, which violations validity.

11.6 Staying in a bivalent configuration

Now start in a failure-free bivalent configuration C with some step $e = (p, m)$ or $e = (p, \perp)$ enabled in C. Let S be the set of configurations reachable from C without doing e or failing any processes, and let $e(S)$ be the set of configurations of the form $C'e$ where C' is in S. (Note that e is always enabled in S, since once enabled the only way to get rid of it is to deliver the message.) We want to show that $e(S)$ contains a failure-free bivalent configuration.

The proof is by contradiction: suppose that $C'e$ is univalent for all C' in S. We will show first that there are C_0 and C_1 in S such that each C_ie is i-valent. To do so, consider any pair of i-valent A_i reachable from C; if A_i is in S, let $C_i = A_i$. If A_i is not in S, let C_i be the last configuration before executing e on the path from C to A_i (C_ie is univalent in this case by assumption).

So now we have C_0e and C_1e with C_ie i-valent in each case. We'll now go hunting for some configuration D in S and step e' such that De is 0-valent but $De'e$ is 1-valent (or vice versa); such a pair exists because S is connected and so some step e' crosses the boundary between the $C'e$ = 0-valent and the $C'e$ = 1-valent regions.

By a case analysis on e and e' we derive a contradiction:

1. Suppose e and e' are steps of different processes p and p'. Let both steps go through in either order. Then $Dee' = De'e$, since in an asynchronous system we can't tell which process received its message first. But De is 0-valent, which implies Dee' is also 0-valent, which contradicts $De'e$ being 1-valent.

2. Now suppose e and e' are steps of the same process p. Again we let both go through in either order. It is not the case now that $Dee' = De'e$, since p knows which step happened first (and may have sent messages telling the other processes). But now we consider some finite sequence of steps $e_1e_2 \ldots e_k$ in which no message sent by p is delivered and some process decides in $Dee_1 \ldots e_k$ (this occurs since the other processes can't distinguish Dee' from the configuration in which p died in D, and so have to decide without waiting for messages from p). This execution fragment is indistinguishable to all processes except p from $De'ee_1 \ldots e_k$, so the deciding process decides the same value i in both executions. But Dee' is 0-valent and $De'e$ is 1-valent, giving a contradiction.

It follows that our assumption was false, and there is some reachable bivalent configuration $C'e$.

Now to construct a fair execution that never decides, we start with a bivalent configuration, choose the oldest enabled action and use the above to make it happen while staying in a bivalent configuration, and repeat.

11.7 Generalization to other models

To apply the argument to another model, the main thing is to replace the definition of a step and the resulting case analysis of 0-valent $De'e$ vs 1-valent Dee' to whatever steps are available in the other model. For example, in asynchronous shared memory, if e and e' are operations on different memory locations, they commute (just like steps of different processes), and if they are operations on the same location, either they commute (e.g. two reads) or only one process can tell whether both happened (e.g. with a write and a read, only the reader knows, and with two writes, only the first writer knows). Killing the witness yields two indistinguishable configurations with different valencies, a contradiction.

We are omitting a lot of details here. See [Lyn96, §12.2] for the real proof, or Loui and Abu-Amara [LAA87] for the generalization to shared memory, or Herlihy [Her91b] for similar arguments for a wide variety of shared-memory primitives. We will see many of these latter arguments in Chapter 18.

Chapter 12

Paxos

The **Paxos** algorithm for consensus in a message-passing system was first described by Lamport in 1990 in a tech report that was widely considered to be a joke (see `http://research.microsoft.com/users/lamport/pubs/pubs.html#lamport-paxos` for Lamport's description of the history). The algorithm was finally published in 1998 [Lam98], and after the algorithm continued to be ignored, Lamport finally gave up and translated the results into readable English [Lam01]. It is now understood to be one of the most efficient practical algorithms for achieving consensus in a message-passing system with failure detectors, mechanisms that allow processes to give up on other stalled processes after some amount of time (which can't be done in a normal asynchronous system because giving up can be made to happen immediately by the adversary).

We will describe the basic Paxos algorithm in §12.1. This is a one-shot version of Paxos that solves a single agreement problem. The version that is more typically used, called **multi-Paxos**, uses repeated executions of the basic Paxos algorithm to implement a replicated state machine; we'll describe this in §12.3.

There are many more variants of Paxos in use. The WikiPedia article on Paxos (`http://en.wikipedia.org/wiki/Paxos_(computer_science)`) gives a remarkably good survey of subsequent developments and applications.

12.1 The Paxos algorithm

The algorithm runs in a message-passing model with asynchrony and fewer than $n/2$ crash failures (but not Byzantine failures, at least in the original algorithm). As always, we want to get agreement, validity, and termination.

The Paxos algorithm itself is mostly concerned with guaranteeing agreement and validity, while allowing for the possibility of termination if there is a long enough interval in which no process restarts the protocol.

Processes are classified as **proposers**, **accepters**, and **learners** (a single process may have all three roles). The idea is that a proposer attempts to ratify a proposed decision value (from an arbitrary input set) by collecting acceptances from a majority of the accepters, and this ratification is observed by the learners. Agreement is enforced by guaranteeing that only one proposal can get the votes of a majority of accepters, and validity follows from only allowing input values to be proposed. The tricky part is ensuring that we don't get deadlock when there are more than two proposals or when some of the processes fail. The intuition behind how this works is that any proposer can effectively restart the protocol by issuing a new proposal (thus dealing with lockups), and there is a procedure to release accepters from their old votes if we can prove that the old votes were for a value that won't be getting a majority any time soon.

To organize this vote-release process, we attach a distinct proposal number to each proposal. The safety properties of the algorithm don't depend on anything but the proposal numbers being distinct, but since higher numbers override lower numbers, to make progress we'll need them to increase over time. The simplest way to do this in practice is to make the proposal number be a timestamp with the proposer's id appended to break ties. We could also have the proposer poll the other processes for the most recent proposal number they've seen and add 1 to it.

The revoting mechanism now works like this: before taking a vote, a proposer tests the waters by sending a $\mathsf{prepare}(n)$ message to all accepters, where n is the proposal number. An accepter responds to this with a promise never to accept any proposal with a number less than n (so that old proposals don't suddenly get ratified) together with the highest-numbered proposal that the accepter has accepted (so that the proposer can substitute this value for its own, in case the previous value was in fact ratified). If the proposer receives a response from a majority of the accepters, the proposer then does a second phase of voting where it sends $\mathsf{accept}(n, v)$ to all accepters and wins if receives a majority of votes.

So for each proposal, the algorithm proceeds as follows:

1. The proposer sends a message $\mathsf{prepare}(n)$ to all accepters. (Sending to only a majority of the accepters is enough, assuming they will all respond.)

2. Each accepter compares n to the highest-numbered proposal for which

it has responded to a prepare message. If n is greater, it responds with $\mathsf{ack}(n, v, n_v)$, where v is the highest-numbered proposal it has accepted and n_v is the number of that proposal (or $\perp$ and 0 if there is no such proposal).

An optimization at this point is to allow the accepter to send back $\mathsf{nack}(n, n')$ where n' is some higher number to let the proposer know that it's doomed and should back off and try again with a higher proposal number. (This keeps a confused proposer who thinks it's the future from locking up the protocol until 2037.)

3. The proposer waits (possibly forever) to receive ack from a majority of accepters. If any ack contained a value, it sets v to the most recent (in proposal number ordering) value that it received. It then sends $\mathsf{accept}(n, v)$ to all accepters (or just a majority). You should think of accept as a demand ("Accept!") rather than acquiescence ("I accept")—the accepters still need to choose whether to accept or not.

4. Upon receiving $\mathsf{accept}(n, v)$, an accepter accepts v unless it has already received $\mathsf{prepare}(n')$ for some $n' > n$. If a majority of acceptors accept the value of a given proposal, that value becomes the decision value of the protocol.

Note that acceptance is a purely local phenomenon; additional messages are needed to detect which if any proposals have been accepted by a majority of accepters. Typically this involves a fourth round, where accepters send $\mathsf{accepted}(n, v)$ to all learners (often just the original proposer).

There is no requirement that only a single proposal is sent out (indeed, if proposers can fail we will need to send out more to jump-start the protocol). The protocol guarantees agreement and validity no matter how many proposers there are and no matter how often they start.

12.2 Informal analysis: how information flows between rounds

Call a **round** the collection of all messages labeled with some particular proposal n. The structure of the algorithm simulates a sequential execution in which higher-numbered rounds follow lower-numbered ones, even though there is no guarantee that this is actually the case in a real execution.

When an acceptor sends $\mathsf{ack}(n, v, n_v)$, it is telling the round-n proposer the last value preceding round n that it accepted. The rule that an acceptor

only acknowledges a proposal higher than any proposal it has previously acknowledged prevents it from sending information "back in time"—the round n_v in an acknowledgment is always less than n. The rule that an acceptor doesn't accept any proposal earlier than a round it has acknowledged means that the value v in an $\mathsf{ack}(n, v, n_v)$ message never goes out of date—there is no possibility that an acceptor might retroactively accept some later value in round n' with $n_v < n' < n$. So the ack message values tell a consistent story about the history of the protocol, even if the rounds execute out of order.

The second trick is to use overlapping majorities to make sure that any value that is accepted is not lost. If the only way to decide on a value in round n is to get a majority of acceptors to accept it, and the only way to make progress in round n' is to get acknowledgments from a majority of acceptors, these two majorities overlap. So in particular the overlapping process reports the round-n proposal value to the proposer in round n', and we can show by induction on n' that this round-n proposal value becomes the proposal value in all subsequent rounds that proceed past the acknowledgment stage. So even though it may not be possible to detect that a decision has been reached in round n (say, because some of the acceptors in the accepting majority die without telling anybody what they did), no later round will be able to choose a different value. This ultimately guarantees agreement.

12.2.1 Example execution

For Paxos to work well, proposal numbers should increase over time. But there is no requirement that proposal numbers are increasing or even that proposals with different proposal numbers don't overlap. When thinking about Paxos, it is easy to make the mistake of ignore cases where proposals are processed concurrently or out of order. In Figure 12.1, we give an example of an execution with three proposals running concurrently.

12.2.2 Safety properties

We now present a more formal analysis of the Paxos protocol. We consider only the safety properties of the protocol, corresponding to validity and agreement. Without additional assumptions, Paxos does *not* guarantee termination.

Call a value *chosen* if it is accepted by a majority of accepters. The safety properties of Paxos are:

- No value is chosen unless it is first proposed. (This gives validity.)

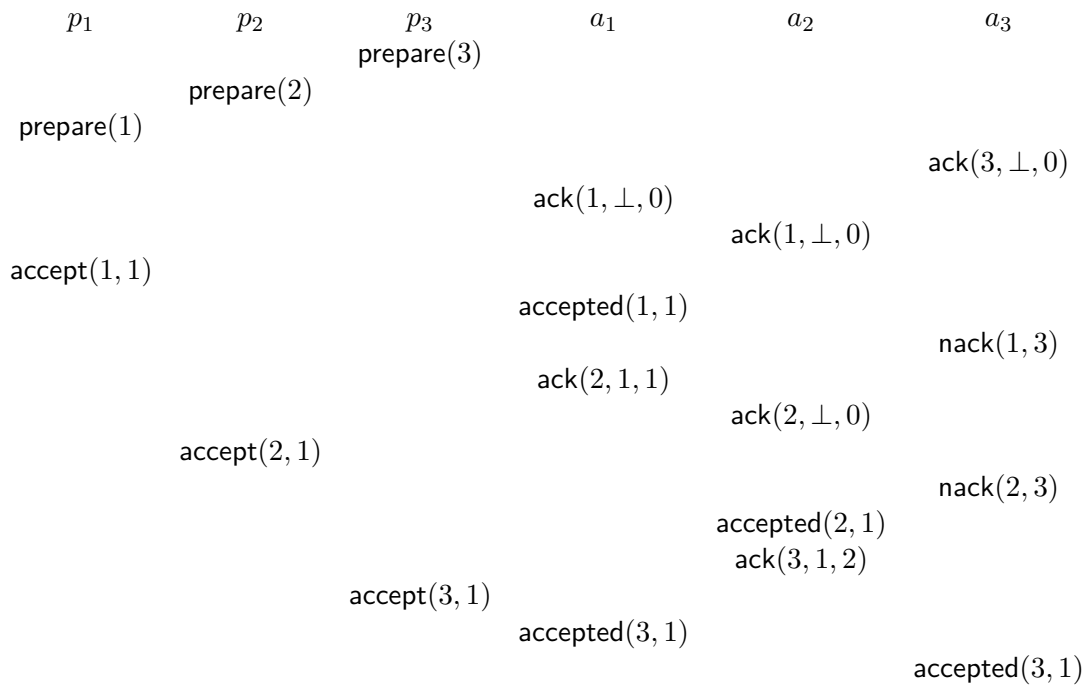

Figure 12.1: Example execution of Paxos. There are three proposers p_1, p_2, and p_3 and three acceptors a_1, a_2, and a_3. Proposer p_1's proposed value 1 is not accepted by a majority of processes in round 1, but it is picked up by proposer p_2 in round 2, and is eventually adopted and accepted in round 3.

- No two distinct values are both chosen. (This gives agreement.)

The first property is immediate from examination of the algorithm: every value propagated through the algorithm is ultimately copy of some proposer's original input.

For the second property, we'll show by induction on proposal number than a value v chosen with proposal number n is the value chosen by any proposer $p_{n'}$ with proposal number n'. There are two things that make this true:

1. Any $\mathsf{ack}(n', v', n_{v'})$ message received by $p_{n'}$ has $n_{v'} < n'$. Proof: immediate from the protocol.

2. If a majority of acceptors accept a proposal with number n, and $p_{n'}$ receives $\mathsf{ack}(n', -, -)$ messages from a majority of acceptors, then $p_{n'}$ receives at least one $\mathsf{ack}(n', v', n_{v'})$ message with $n' \geq n$. Proof: Let S be the set of processes that issue $\mathsf{accepted}(n, v)$ and let T be the set of processes that send $\mathsf{ack}(n', -, -)$ to p'. Because S and T are both majorities, there is at least one acceptor a in $S \cap T$. Suppose $p_{n'}$ receives $\mathsf{ack}(n, v'', n'')$ from a. If $n'' < n$, then at the time a sends its $\mathsf{ack}(n, v'', n'')$ message, it has not yet accepted a proposal with number n. But then when it does receive $\mathsf{accept}(n, v)$, it rejects it. This contradicts $a \in S$.

These two properties together imply that $p_{n'}$ receives at least one $\mathsf{ack}(n, v'', n'')$ with $n \leq n'' < n'$ and no such messages with $n'' < n$. So the maximum proposal number it sees is n'' where $n \leq n'' < n$. By the induction hypothesis, the corresponding value is v. It follows that $p_{n'}$ also chooses v.

12.2.3 Learning the results

Somebody has to find out that a majority accepted a proposal in order to get a decision value out. The usual way to do this is to have a fourth round of messages where the accepters send $\mathsf{accepted}(v, n)$ to some designated learner (usually just the original proposer), which can then notify everybody else if it doesn't fail first. If the designated learner does fail first, we can restart by issuing a new proposal (which will get replaced by the previous successful proposal because of the safety properties).

12.2.4 Liveness properties

We'd like the protocol to terminate eventually. Suppose there is a single proposer, and that it survives long enough to collect a majority of acks and to send out accepts to a majority of the accepters. If everybody else cooperates, we get termination in 4 message delays, including the time for the learners to detect acceptance.

If there are multiple proposers, then they can step on each other. For example, it's enough to have two carefully-synchronized proposers alternate sending out prepare messages to prevent any accepter from every accepting (since an accepter promises not to accept accept(n, v) once it has responded to prepare$(n + 1)$). The solution is to ensure that there is eventually some interval during which there is exactly one proposer who doesn't fail. One way to do this is to use exponential random backoff (as popularized by Ethernet): when a proposer decides it's not going to win a round (e.g. by receiving a nack or by waiting long enough to realize it won't be getting any more acks soon), it picks some increasingly large random delay before starting a new round; thus two or more will eventually start far enough apart in time that one will get done without interference.

A more abstract solution is to assume some sort of weak leader election mechanism, which tells each accepter who the "legitimate" proposer is at each time. The accepters then discard messages from illegitimate proposers, which prevents conflict at the cost of possibly preventing progress. Progress is however obtained if the mechanism eventually reaches a state where a majority of the accepters bow to the same non-faulty proposer long enough for the proposal to go through.

Such a weak leader election method is an example of a more general class of mechanisms known as **failure detectors**, in which each process gets hints about what other processes are faulty that eventually converge to reality. The particular failure detector in this case is known as the Ω failure detector; there are other still weaker ones that we will talk about later that can also be used to solve consensus. We will discuss failure detectors in detail in Chapter 13.

12.3 Replicated state machines and multi-Paxos

The most common practical use of Paxos is to implement a **replicated state machine** [Lam78]. The idea is to maintain many copies of some data structure, each on a separate machine, and guarantee that each copy (or **replica**) stays in sync with all the others as new operations are applied

to them. This requires some mechanism to ensure that all the different replicas apply the same sequence of operations, or in other words that the machines that hold the replicas solve a sequence of agreement problems to agree on these operations. The payoff is that the state of the data structure survives the failure of some of the machines, without having to copy the entire structure every time it changes.

Paxos works well as the agreement mechanism because the proposer can also act as a dispatcher for operations. Instead of having to broadcast an operation to all replicas, a user can contact the current active proposer, and the proposer can include the operation in its next proposal. If the proposer doesn't change very often, a further optimization allows skipping the prepare and ack messages in between agreement protocols for consecutive operations. This reduces the time to certify each operation to a single round-trip for the accept and accepted messages.

In order to make this work, we need to distinguish between successive proposals by the same proposer, and "new" proposals that change the proposer in addition to reaching agreement on some value. This is done by splitting the proposal number into a major and minor number, with proposals ordered lexicographically. A proposer that wins $\langle x, 0\rangle$ is allowed to make further proposals numbered $\langle x, 1\rangle, \langle x, 2\rangle$, etc. But a different proposer will need to increment x.

Lamport calls this optimization Paxos in [Lam01]; other authors have called it **multi-Paxos** to distinguish it from the basic Paxos algorithm.