Chapter 11

Impossibility of asynchronous agreement

There's an easy argument that says that you can't do most things in an asynchronous message-passing system with $n/2$ crash failures: partition the processes into two subsets S and T of size $n/2$ each, and allow no messages between the two sides of the partition for some long period of time. Since the processes in each side can't distinguish between the other side being slow and being dead, eventually each has to take action on their own. For many problems, we can show that this leads to a bad configuration. For example, for agreement, we can supply each side of the partition with a different common input value, forcing disagreement because of validity. We can then satisfy the fairness condition that says all messages are eventually delivered by delivering the delayed messages across the partition, but it's too late for the protocol.

The Fischer-Lynch-Paterson (FLP) result [FLP85] says something much stronger: you can't do agreement in an asynchronous message-passing system if even *one* crash failure is allowed.[1] 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 operations

[1] Unless you augment the basic model in some way, say by adding randomization (Chapter 23) or failure detectors (Chapter 13).

like atomic snapshots[2]) 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.

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

[2] Chapter 19.

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 $e_1 e_2 \dots e_k$ in which no message sent by p is delivered and some process decides in $Dee_1 \dots e_k$ (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 $De'ee_1 \dots e_k$, 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.