Note: These notes are based on material from James Aspnes (see reference below)

1 The Consensus Problem

In the *Consensus Problem*, every process starts with a single input bit. We require that all non-faulty processes terminate and also:

- **Agreement:** All non-faulty processes decide the same value
- **Validity:** The value decided must equal the input bit of some non-faulty process

2 Definitions and Model

Definition 1. *A* step *is either a pair (p, m) where process p receives message m or a pair $(p, \bot)$ where p receives nothing and performs at least one send. We say that a step (p, m) is* enabled *when a message m has been sent to process p, and the step (p, m) is* applied *when process p receives the message m. A step $(p, \bot)$ is enabled when process p has reached a step in its algorithm to send at least one message, and $(p, \bot)$ is applied when p sends at least one message.*

Definition 2. *A* configuration *consists of the states of all processors, and all events that have been enabled but have not yet been applied by the scheduler.*

Our model assumes *fairness:* if (p, m) or $(p, \bot)$ is continuously enabled it eventually is applied. Since messages are never lost, once (p, m) is enabled in some configuration C, it is enabled in all successor configurations until if eventually happens; same for $(p, \bot)$. Then, any non-faulty process eventually performs any enabled step.

Our model is *asynchronous* in the sense that *all* we assume is fairness. The adversary always gets to choose which enabled step to apply, provided that it eventually applies each enabled step.

3 FLP Result

The FLP theorem statement is simple and stark.

Theorem 1. *There is no deterministic algorithm for consensus in the asynchronous model that tolerates even a single fault.*

"If you encounter someone posturing as an expert in consensus, I enourage you to ask them if they know the statement of the FLP impossibility theorem. If you really want to be hostile, ask them if they know anything about the proof (which as we'll see is not so easy" - Tim Roughgarden

4 Proof Overview

For configuration C and enabled step e, we will write the new configuration reached by applying step e as Ce.

Let a *trace* be a sequence of steps. We will write CT as the new configuration reached applying all steps in T to C.

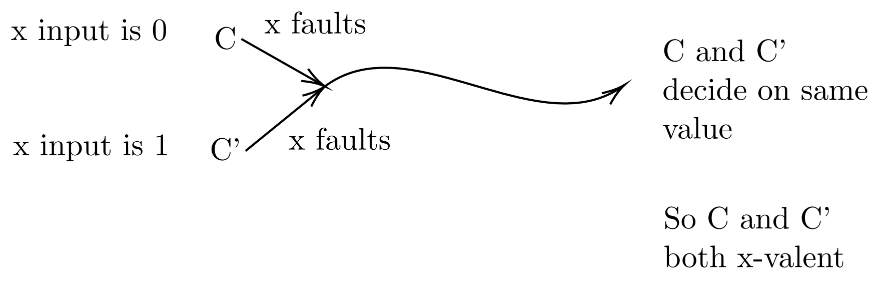

Figure 1. Assume initial configurations C and C' are univalent and differ only in the input of processor x.

Definition 3. *A configuration C is* bivalent *if there exist traces T_1 and T_2 such that CT_1 has all processes output* 0 *and CT_2 has all processes output* 1*. A configuration that is* univalent *if it is not bivalent. Specifically, it is either* 0*-valent if it leads to outputs of* 0 *and* 1*-valent otherwise.*

We will use the fact that the successor of any x-valent configuration is also x-valent.

Our general approach will be to show that we - remember "we" are the adversarial scheduler - can keep any protocol in a bivalent state forever. This implies that we can keep any protocol from ever deciding. A complication is that due to the fairness constraint in our model, we must also allow every message to eventually be delivered; i.e. **every enabled step must eventually be applied.** We'll ensure this below by continually ensuring that we apply the oldest enabled step.

5 Initial Bivalent Configuration

Lemma 1. *There exists a bivalent initial configuration.*

Proof: Assume not. Then all initial configurations are univalent. Consider any two such configurations C and C' that differ only in the input of a single process x; See Figure 1. The adversary can make process x fault immediately and so ensure that configurations C and C' both decide on the same value. Hence, C and C' must have the same valency.

But, we can change any initial configuration to any other initial configuration by traversing through configurations where only a single process's input is changed. So, *all* initial configurations must have exactly the same valency. But, this contradicts the assumption of validity for consensus. Hence, our initial assumption was wrong and there must be some bivalent initial configuration. □

We call a configuration *faultless* if no fault has occurred in the configuration.

Lemma 2. *Let C be any faultless, bivalent configuration, and let e be any enabled step in C. Then, from C we can reach some other faultless, bivalent configuration in which step e has been applied.*

Proof: Let S be the set of faultless configurations reachable from C in which step e has not been applied (Figure 2, above the vertical line)

Assume, by way of contradiction, that there is no configuration C' in S such that $C'e$ is bivalent. Then, since C is bivalent, there must be a pair of configurations C_0 and C_1 in S such that C_0e is 0-valent and C_1e is 1-valent (Figure 2). Since all configurations in S are reachable from C, and C is univalent, there must be some configuration D and step e' such that De is 0-valent and $(De')e$ is 1-valent (or vice-versa).

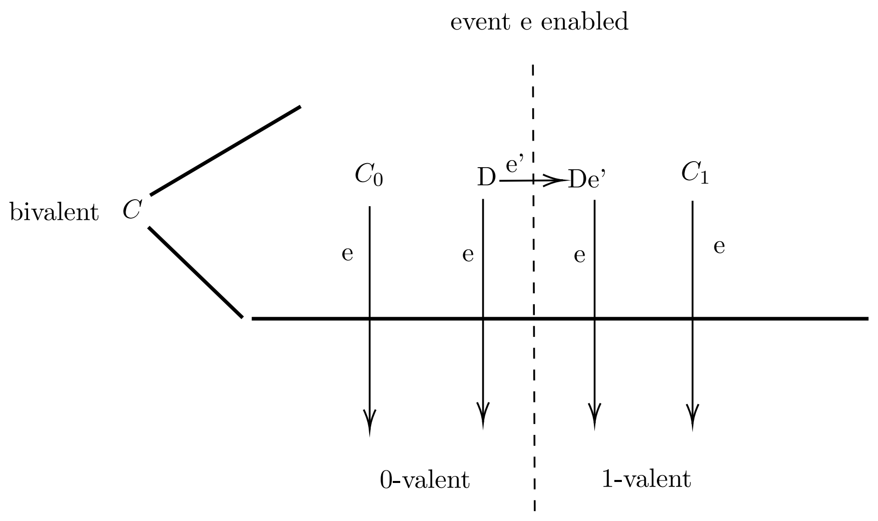

Figure 2. The configuration D and step e'

There are two cases:

CASE 1: e and e' are steps of different processors p and p'. In this case, $Dee' = De'e$, since in an asynchronous system, no one can tell which process of the two had its step applied first. Since Dee' is 0-valent, Dee' is also 0-valent. But, $De'e$ is 1-valent. This is a contradiction!

CASE 2: e and e' are steps of the same processor p. Consider some finite sequence of steps $\sigma = e_1e_2 \ldots e_k$ from D after steps e' and e occur (in either order), in which no message from p is delivered and some processor decides. Such a sequence must exist since p could fault. But then the valency of configuration $Dee'\sigma$ equals the valency of configuration $De'e\sigma$, since the other processors never learn the order of e and e'. But, Dee' is 0-valent and $De'e$ is 1 valent. This is a contradiction!

Since we have a contradiction in both cases, there must be some configuration C' in S such that $C'e$ is bivalent. □

Theorem 2. *No algorithm can solve Consensus in the asynchronous model and tolerate* 1 *fault.*

Proof: By Lemma 1, there is an initial bivalent configuration. This is where the adversary starts the algorithm.

We will show that the adversary can keep the algorithm in faultless, bivalent configurations indefinitely, while still ensuring fairness: every enabled step is eventually activated. This shows that any algorithm can be caused to never terminate.

To show this, consider some faultless, bivalent configuration, C that the algorithm is currently in, and let e be the oldest, enabled step in C. By Lemma 2, there is some configuration C' that is reachable from C such that $C'e$ is a faultless, bivalent configuration. So the adversary simply schedules steps in such a way that we first reach C' and then enable step e. We can repeat this process indefinitely, while ensuring that every enabled step is eventually applied.

□

6 References

- Aspnes Notes: https://www.cs.yale.edu/homes/aspnes/pinewiki/FischerLynchPaterson.html