Chapter 8

Coordinated attack

Last updated 2025. Some material may be out of date.

(See also [Lyn96, §5.1].)

The **Two Generals** problem was the first widely-known distributed consensus problem, described in 1978 by Jim Gray [Gra78, §5.8.3.3.1], although the same problem previously appeared under a different name [AEH75].

The setup of the problem is that we have two generals on opposite sides of an enemy army, who must choose whether to attack the army or retreat. If only one general attacks, his troops will be slaughtered. So the generals need to reach agreement on their strategy.

To complicate matters, the generals can only communicate by sending messages by (unreliable) carrier pigeon. We also suppose that at some point each general must make an irrevocable decision to attack or retreat. The interesting property of the problem is that if carrier pigeons can become lost, there is no protocol that guarantees agreement in all cases unless the outcome is predetermined (e.g., the generals always attack no matter what happens). The essential idea of the proof is that any protocol that does guarantee agreement can be shortened by deleting the last message; iterating this process eventually leaves a protocol with no messages.

Adding more generals turns this into the **coordinated attack** problem, a variant of **consensus**, but it doesn't make things any easier.

8.1 Formal description

To formalize this intuition, suppose that we have $n \geq 2$ generals in a synchronous system with unreliable channels—the set of messages received in round $i + 1$ is always a subset of the set sent in round i, but it may be a proper subset (even the empty set). Each general starts with an input 0 (retreat) or 1 (attack) and must output 0 or 1 after some bounded number of rounds. The requirements for the protocol are that, in all executions:

Agreement All processes output the same decision (0 or 1).

Validity If all processes have the same input x, and no messages are lost, all processes produce output x. (If processes start with different inputs or one or more messages are lost, processes can output 0 or 1 as long as they all agree.)

Termination All processes terminate in a bounded number of rounds.[1]

Sadly, there is not protocol that satisfies all three conditions. We show this in the next section.

8.2 Impossibility proof

To show coordinated attack is impossible,[2] we use an **indistinguishability proof**.

The key steps of an indistinguishability proof usually look like this:

- Show that execution A is **indistinguishable** from execution B for some process p, meaning that p sees the same things (messages or operation results) in both executions.
- Observe that if A is indistinguishable from B for p, then because p can't tell which of these two possible worlds it is in, it returns the same output in both.

So far, pretty dull. But now let's consider a chain of hypothetical executions $A = A_0 A_1 \ldots A_k = B$, where each A_i is indistinguishable from A_{i+1} for some process p_i. Suppose also that we are trying to solve an

[1] **Bounded** means that there is a fixed upper bound on the length of any execution. We could also demand merely that all processes terminate in a *finite* number of rounds. In general, finite is a weaker requirement than bounded, but if the number of possible outcomes at each step is finite (as they are in this case), they're equivalent. The reason is that if we build a tree of all configurations, each configuration has only finitely many successors, and the length of each path is finite, then **König's lemma** (see `http://en.wikipedia.org/wiki/Konig's_lemma`) says that there are only finitely many paths. So we can take the length of the longest of these paths as our fixed bound. [BG97, Lemma 3.1]

[2] Without making additional assumptions, always a caveat when discussing impossibility.

agreement task, where every process must output the same value. Then since p_i outputs the same value in A_i and A_{i+1}, every process outputs the same value in A_i and A_{i+1}. By induction on k, every process outputs the same value in A and B, even though A and B may be very different executions.

This gives us a tool for proving impossibility results for agreement: show that there is a path of indistinguishable executions between two executions that are supposed to produce different output. Another way to picture this: consider a graph whose nodes are all possible executions with an edge between any two indistinguishable executions; then the set of output-0 executions can't be adjacent to the set of output-1 executions. If we prove the graph is connected, we prove the output is the same for all executions.

For coordinated attack, we will show that no protocol satisfies all of agreement, validity, and termination using an indistinguishability argument. The key idea is to construct a path between the all-0-input and all-1-input executions with no message loss via intermediate executions that are indistinguishable to at least one process.

Let's start with $A = A_0$ being an execution in which all inputs are 1 and all messages are delivered. We'll build executions A_1, A_2, etc., by pruning messages. Consider A_i and let m be some message that is delivered in the last round in which any message is delivered. Construct A_{i+1} by not delivering m. Observe that while A_i is distinguishable from A_{i+1} by the recipient of m, on the assumption that $n \geq 2$ there is some other process that can't tell whether m was delivered or not (the recipient can't let that other process know, because no subsequent message it sends are delivered in either execution). Continue until we reach an execution A_k in which all inputs are 1 and no messages are sent. Next, let A_{k+1} through A_{k+n} be obtained by changing one input at a time from 1 to 0; each such execution is indistinguishable from its predecessor by any process whose input didn't change. Finally, construct A_{k+n} through $A_{k+n+k'}$ by adding back messages in the reverse process used for A_0 through A_k; note that this might not result in exactly k new messages, because the number of messages might depend on the inputs. This gets us to an execution $A_{k+n+k'}$ in which all processes have input 0 and no messages are lost. If agreement holds, then the indistinguishability of adjacent executions to some process means that the common output in A_0 is the same as in $A_{k+n+k'}$. But validity requires that A_0 outputs 1 and $A_{k+n+k'}$ outputs 0: so either agreement or validity is violated in some execution.

8.3 Randomized coordinated attack

So we now know that we can't solve the coordinated attack problem. But maybe we want to solve it anyway. The solution is to change the problem.

Randomized coordinated attack is like standard coordinated attack, but with less coordination. Specifically, we'll allow the processes to flip coins to decide what to do, and assume that the communication pattern (which messages get delivered in each round) is fixed and independent of the coin-flips. This corresponds to assuming an **oblivious adversary** that can't see what is going on at all or perhaps a **content-oblivious adversary** that can only see where messages are being sent but not the contents of the messages. We'll also relax the agreement property to only hold with some high probability:

Randomized agreement For any adversary A, the probability that some process decides 0 and some other process decides 1 given A is at most ϵ.

Validity and termination are as before.

8.3.1 An algorithm

Here's an algorithm that gives $\epsilon = 1/r$. (See [Lyn96, §5.2.2] for details or [VL92] for the original version.) A simplifying assumption is that network is complete, although a strongly-connected network with r greater than or equal to the diameter also works.

- First part: tracking information levels
  - Each process tracks its "information level," initially 0. The state of a process consists of a vector of (input, information-level) pairs for all processes in the system. Initially this is (my-input, 0) for itself and $(\perp, -1)$ for everybody else.
  - Every process sends its entire state to every other process in every round.
  - Upon receiving a message m, process i stores any inputs carried in m and, for each process j, sets $\mathsf{level}_i[j]$ to $\max(\mathsf{level}_i[j], \mathsf{level}_m[j])$. It then sets its own information level to $\min_j(\mathsf{level}_i[j]) + 1$.
- Second part: deciding the output
  - Process 1 chooses a random key value uniformly in the range $[1, r]$.

- This key is distributed along with $\mathsf{level}_i[1]$, so that every process with $\mathsf{level}_i[1] \geq 0$ knows the key.
- A process decides 1 at round r if and only if it knows the key, its information level is greater than or equal to the key, and all inputs are 1.

8.3.2 Why it works

Termination Immediate from the algorithm.

Validity

- If all inputs are 0, no process sees all 1 inputs (technically requires an invariant that processes' non-null views are consistent with the inputs, but that's not hard to prove.)
- If all inputs are 1 and no messages are lost, then the information level of each process after k rounds is k (prove by induction) and all processes learn the key and all inputs (immediate from first round). So all processes decide 1.

Randomized Agreement

- First prove a lemma: Define $\mathsf{level}_i^t[k]$ to be the value of $\mathsf{level}_i[k]$ after t rounds. Then for all i, j, k, t, (1) $\mathsf{level}_i[j]^t \leq \mathsf{level}_j[j]^{t-1}$ and (2) $\left|\mathsf{level}_i[k]^t - \mathsf{level}_j[k]^t\right| \leq 1$. As always, the proof is by induction on rounds. Part (1) is easy and boring so we'll skip it. For part (2), we have:
  - After 0 rounds, $\mathsf{level}_i^0[k] = \mathsf{level}_j^0[k] = -1$ if neither i nor j equals k; if one of them is k, we have $\mathsf{level}_k^0[k] = 0$, which is still close enough.
  - After t rounds, consider $\mathsf{level}_i^t[k] - \mathsf{level}_i^{t-1}[k]$ and similarly $\mathsf{level}_j^t[k] - \mathsf{level}_j^{t-1}[k]$. It's not hard to show that each can jump by at most 1. If both deltas are $+1$ or both are 0, there's no change in the difference in views and we win from the induction hypothesis. So the interesting case is when $\mathsf{level}_i[k]$ stays the same and $\mathsf{level}_j[k]$ increases or vice versa.
  - There are two ways for $\mathsf{level}_j[k]$ to increase:
    * If $j \neq k$, then j received a message from some j' with $\mathsf{level}_{j'}^{t-1}[k] > \mathsf{level}_j^{t-1}[k]$. From the induction hypothesis, $\mathsf{level}_{j'}^{t-1}[k] \leq \mathsf{level}_i^{t-1}[k] + 1 = \mathsf{level}_i^t[k]$. So we are happy.
    * If $j = k$, then j has $\mathsf{level}_j^t[j] = 1 + \min_{k \neq j} \mathsf{level}_j^t[k] \leq 1 + \mathsf{level}_j^t[i] \leq 1 + \mathsf{level}_i^t[i]$. Again we are happy.

- Note that in the preceding, the key value didn't figure in; so everybody's level at round r is independent of the key.
- So now we have that $\mathsf{level}^r_i[i]$ is in $\{\ell, \ell+1\}$, where ℓ is some fixed value uncorrelated with the key. The only way to get some process to decide 1 while others decide 0 is if $\ell + 1 \geq \mathsf{key}$ but $\ell < \mathsf{key}$. (If $\ell = 0$, a process at this level doesn't know key, but it can still reason that $0 < \mathsf{key}$ since key is in $[1, r]$.) This can only occur if $\mathsf{key} = \ell + 1$, which occurs with probability at most $1/r$ since key was chosen uniformly.

8.3.3 Almost-matching lower bound

The bound on the probability of disagreement in the previous algorithm is almost tight. Varghese and Lynch [VL92] show that no synchronous algorithm can get a probability of disagreement less than $\frac{1}{r+1}$, using a stronger validity condition that requires that the processes output 0 if any input is 0. This is a natural assumption for database commit, where we don't want to commit if any process wants to abort. We restate their result below:

Theorem 8.3.1. *For any synchronous algorithm for randomized coordinated attack that runs in r rounds that satisfies the additional condition that all non-faulty processes decide* 0 *if any input is* 0*,* $\Pr[\textit{disagreement}] \geq 1/(r+1)$.

Proof. Let ϵ be the bound on the probability of disagreement. Define $\mathsf{level}^t_i[k]$ as in the previous algorithm (whatever the real algorithm is doing). We'll show $\Pr[i \text{ decides } 1] \leq \epsilon \cdot (\mathsf{level}^r_i[i] + 1)$, by induction on $\mathsf{level}^r_i[i]$.

- If $\mathsf{level}^r_i[i] = 0$, the real execution is indistinguishable (to i) from an execution in which some other process j starts with 0 and receives no messages at all. In that execution, j must decide 0 or risk violating the strong validity assumption. So i decides 1 with probability at most ϵ (from the disagreement bound).

- If $\mathsf{level}^r_i[i] = k > 0$, the real execution is indistinguishable (to i) from an execution in which some other process j only reaches level $k-1$ and thereafter receives no messages. From the induction hypothesis, $\Pr[j \text{ decides } 1] \leq \epsilon k$ in that pruned execution, and so $\Pr[i \text{ decides } 1] \leq \epsilon(k+1)$ in the pruned execution. But by indistinguishability, we also have $\Pr[i \text{ decides } 1] \leq \epsilon(k+1)$ in the original execution.

Now observe that in the all-1 input execution with no messages lost, $\mathsf{level}^r_i[i] = r$ and $\Pr[i \text{ decides } 1] = 1$ (by validity). So $1 \leq \epsilon(r+1)$, which implies $\epsilon \geq 1/(r+1)$. ☐

Chapter 9

Synchronous agreement

Last updated 2025. Some material may be out of date.

Here we'll consider synchronous agreement algorithm with stopping failures, where a process stops dead at some point, sending and receiving no further messages. We'll also consider Byzantine failures, where a process deviates from its programming by sending arbitrary messages, but mostly just to see how crash-failure algorithms hold up; for algorithms designed specifically for a Byzantine model, see Chapter 10.

If the model has communication failures instead, we have the coordinated attack problem from Chapter 8.

9.1 Problem definition

We use the usual synchronous model with n processes with binary inputs and binary outputs. Up to f processes may fail at some point; when a process fails, one or one or more of its outgoing messages are lost in the round of failure and all outgoing messages are lost thereafter.

There are two variants on the problem, depending on whether we want a useful algorithm (and so want strong conditions to make our algorithm more useful) or a lower bound (and so want weak conditions to make our lower bound more general). For algorithms, we will ask for these conditions to hold:

Agreement All non-faulty processes decide the same value.

Validity If all processes start with the same input, all non-faulty processes decide it.

Termination All non-faulty processes eventually decide.

For lower bounds, we'll replace validity with **non-triviality** (often called validity in the literature):

Non-triviality There exist failure-free executions A and B that produce different outputs.

Non-triviality follows from validity but doesn't imply validity; for example, a non-trivial algorithm might have the property that if all non-faulty processes start with the same input, they all decide something else.

In §9.2, we'll show that a simple algorithm gives agreement, termination, and validity with f failures using $f + 1$ rounds. We'll then show in §9.3 that non-triviality, agreement, and termination imply that $f + 1$ rounds is the best possible. In Chapter 10, we'll show that the agreement is still possible in $f + 1$ rounds even if faulty processes can send arbitrary messages instead of just crashing, but only if the number of faulty processes is strictly less than $n/3$.

9.2 Solution using flooding

The flooding algorithm, due to Dolev and Strong [DS83] gives a straightforward solution to synchronous agreement for the crash failure case. It runs in $f + 1$ rounds assuming f crash failures. The algorithm given here is a gross simplification of Dolev and Strong's original algorithm, which solves the harder problem of authenticated Byzantine agreement. (This algorithm is also described in more detail in [AW04, §5.1.3] or [Lyn96, §6.2.1].)

Each process keeps a set of (process, input) pairs, initially just $\{(\mathsf{myId}, \mathsf{myInput})\}$. At round r, I broadcast my set to everybody and take the union of my set and all sets I receive. At round $f + 1$, I decide on $f(S)$, where f is some fixed function from sets of process-input pairs to outputs that picks some input in S: for example, f might take the input with the smallest process-id attached to it, take the max of all known input values, or take the majority of all known input values.

Lemma 9.2.1. *After $f + 1$ rounds, all non-faulty processes have the same set.*

Proof. Let S_i^r be the set stored by process i after r rounds. What we'll really show is that if there are no failures in round k, then $S_i^r = S_j^r = S_i^{k+1}$ for all i, j, and $r > k$. To show this, observe that no faults in round k means that all processes that are still alive at the start of round k send their message to all other processes. Let L be the set of live processes in round k. At the end of round k, for i in L we have $S_i^{k+1} = \bigcup_{j \in L} S_j^k = S$. Now we'll consider some round $r = k + 1 + m$ and show by induction on m that $S_i^{k+m} = S$; we already did $m = 0$, so for larger m notice that all messages are equal to S and so S_i^{k+1+m} is the union of a whole bunch of S's. So in particular we have $S_i^{f+1} = S$ (since some failure-free round occurred in the preceding $f + 1$ rounds) and everybody decides the same value $f(S)$. $\square$

9.2.1 Authenticated version

Flooding depends on being able to trust second-hand descriptions of values; it may be that process 1 fails in round 0 so that only process 2 learns its input. If process 2 can suddenly tell 3 (but nobody else) about the input in round $f + 1$—or worse, tell a different value to 3 and 4—then we may get disagreement.

Usually we assume that we don't have access to cryptography, but if we include an authentication mechanism that allows processes to attach unforgeable signatures to messages, then the full version of the Dolev-Strong algorithm solves agreement in $f + 1$ even with f **Byzantine** faults, where a process can send any messages it likes regardless of the protocol. The idea is that instead of sending around unauthenticated input values, I send around input values that are authenticated by a sequence of signatures, one for each process that forwarded it. So a value v_1 that started as the input to process p_1 and reached me via processes p_2 and p_3 might arrive in a message as $\langle v_1, 123, S_3(S_2(S_1(v_1)))\rangle$, giving the value, the path it reached me by, and a nested sequence of signatures allowing me to verify that it did in fact travel this path.

To avoid mischief, a process will accept in round r only a message that appears to have traveled a path involving $f+1$ processes, and will only resend values it accepts. We can limit message complexity by having each process resend only the first copy of each value it accepts, and only to processes that are not already listed in the history.

We now have the property that any value a non-faulty process accepts in round $f+1$ passed through $f+1$ processes, including at least one non-faulty process. That non-faulty process will have forwarded it to all non-faulty processes. If a process accepts a value earlier than round $f + 1$, then it forwards it itself. In either case, if you and I are both non-faulty, then I know that my eventual set S is a subset of yours. Since this holds in reverse as well, my S equals your S' and so we decide the same value $f(S) = f(S')$.

The intuition here is that if a Byzantine process can be forced to show its work, Byzantine failures essentially reduce to omission failures, since a non-faulty process can discard any incoming messages that are obviously bogus. For the most part we will not assume that we have the tools to do this, and that catching Byzantine processes will require more careful protocols.

9.3 Lower bound on rounds

Here we show that synchronous agreement requires at least $f + 1$ rounds if f processes can fail. This proof is modeled on the one in [Lyn96, §6.7] and works backwards from the final state; for a proof of the same result that works in the opposite direction, see [AW04, §5.1.4]. The original result (stated for Byzantine failures) is due to Dolev and Strong [DS83], based on a more complicated proof due to Fischer and Lynch [FL82]; see the chapter notes for Chapter 5 of [AW04] for more discussion of the history.

Note that unlike the algorithms in the preceding and following sections, which provide validity, the lower bound applies even if we only demand non-triviality.

Like the similar proof for coordinated attack (§8.2), the proof uses an indistinguishability argument. But we have to construct a more complicated chain of intermediate executions.

A **crash failure** at process i means that (a) in some round r, some or all of the messages sent by i are not delivered, and (b) in subsequent rounds, no messages sent by i are delivered. The intuition is that i keels over dead in the middle of generating its outgoing messages for a round. Otherwise i behaves perfectly correctly. A process that crashes at some point during an execution is called **faulty**

We will show that if up to f processes can crash, and there are at least $f + 2$ processes,[1] then at least $f + 1$ rounds are needed (in some execution) for any algorithm that satisfies agreement, termination, and non-triviality. In particular, we will show that if all executions run in f or fewer rounds, then the indistinguishability graph is connected; this implies non-triviality doesn't hold, because (as in §8.2), two adjacent states must decide the same

[1]With only $f + 1$ processes, we can solve agreement in f rounds using flooding. The idea is that either (a) at most $f - 1$ processes crash, in which case the flooding algorithm guarantees agreement; or (b) exactly f processes crash, in which case the one remaining non-faulty process agrees with itself. So $f + 2$ processes are needed for the lower bound to work, and we should be suspicious of any lower bound proof that does not use this assumption.

value because of the agreement property.[2]

Now for the proof. To simplify the argument, let's assume that all executions terminate in exactly f rounds (we can always have processes send pointless chitchat to pad out short executions) and that every processes sends a message to every other process in every round where it has not crashed (more pointless chitchat). Formally, this means we have a sequence of rounds $0, 1, 2, \ldots, f-1$ where each process sends a message to every other process (assuming no crashes), and a final round f where all processes decide on a value (without sending any additional messages).

We now want to take any two executions A and B and show that both produce the same output. To do this, we'll transform A's inputs into B's inputs one process at a time, crashing processes to hide the changes. The problem is that just crashing the process whose input changed might change the decision value—so we have to crash later witnesses carefully to maintain indistinguishability all the way across the chain.

Let's say that a process p **crashes fully** in round r if it crashes in round r and no round-r messages from p are delivered. The **communication pattern** of an execution describes which messages are delivered between processes without considering their contents—in particular, it tells us which processes crash and what other processes they manage to talk to in the round in which they crash.

With these definitions, we can state and prove a rather complicated induction hypothesis:

Lemma 9.3.1. *For any f-round protocol with $n \geq f+2$ processes permitting up to f crash failures; any process p; and any execution A in which at most one process crashes per round in rounds $0 \ldots r-1$, p crashes fully in round $r+1$, and no other processes crash; there is a sequence of executions $A = A_0 A_1 \ldots A_k$ such that each A_i is indistinguishable from A_{i+1} by some process, each A_i has at most one crash per round, and the communication pattern in A_k is identical to A except that p crashes fully in round r.*

Proof. By induction on $f - r$. If $r = f$, we just crash p in round r and nobody else notices. For $r < f$, first crash p in round r instead of $r+1$, but deliver all of its round-r messages anyway (this is needed to make space for some other process to crash in round $r+1$). Then choose some message m sent by p in round r, and let p' be the recipient of m. We will show that we can produce a chain of indistinguishable executions between any execution in which m is delivered and the corresponding execution in which it is not.

[2]The same argument works with even a weaker version of non-triviality that omits the requirement that A and B are failure-free, but we'll keep things simple.

If $r = f - 1$, this is easy; only p' knows whether m has been delivered, and since $n \geq f + 2$, there exists another non-faulty p'' that can't distinguish between these two executions, since p' sends no messages in round f or later. If $r < f - 1$, we have to make sure p' doesn't tell anybody about the missing message.

By the induction hypothesis, there is a sequence of executions starting with A and ending with p' crashing fully in round $r + 1$, such that each execution is indistinguishable from its predecessor. Now construct the sequence

$$\begin{aligned} A &\to (A \text{ with } p' \text{ crashing fully in } r+1) \\ &\to (A \text{ with } p' \text{ crashing fully in } r+1 \text{ and } m \text{ lost}) \\ &\to (A \text{ with } m \text{ lost and } p' \text{ not crashing}). \end{aligned}$$

The first and last step apply the induction hypothesis; the middle one yields indistinguishable executions since only p' can tell the difference between m arriving or not and its lips are sealed.

We've shown that we can remove one message through a sequence of executions where each pair of adjacent executions is indistinguishable to some process. Now paste together $n - 1$ such sequences (one per message) to prove the lemma. □

The rest of the proof: Crash some process fully in round 0 and then change its input. Repeat until all inputs are changed.

9.4 Variants

So far we have described **binary consensus**, since all inputs are 0 or 1. We can also allow larger input sets. With crash failures, this allows a stronger validity condition: the output must be equal to some non-faulty process's input. It's not hard to see that Dolev-Strong (§9.2) gives this stronger condition.

Chapter 10

Byzantine agreement

Last updated 2025. Some material may be out of date.

Like synchronous agreement (as in Chapter 9) except that we replace crash failures with **Byzantine failures**, where a faulty process can ignore its programming and send any messages it likes. Since we are operating under a universal quantifier, this includes the case where the Byzantine processes appear to be colluding with each other under the control of a centralized adversary.

10.1 Lower bounds

We'll start by looking at lower bounds.

10.1.1 Minimum number of rounds

We've already seen an $f+1$ lower bound on rounds for crash failures (see §9.3). This lower bound applies *a fortiori* to Byzantine failures, since Byzantine failures can simulate crash failures.

10.1.2 Minimum number of processes

We can also show that we need $n > 3f$ processes. For $n = 3$ and $f = 1$ the intuition is that Byzantine B can play non-faulty A and C off against each other, telling A that C is Byzantine and C that A is Byzantine. Since A is telling C the same thing about B that B is saying about A, C can't tell the difference and doesn't know who to believe. Unfortunately, this tragic soap

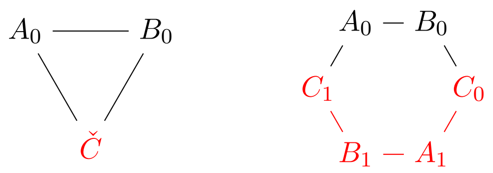

Figure 10.1: Three-process vs. six-process execution in Byzantine agreement lower bound. Processes A_0 and B_0 in right-hand execution receive same messages as in left-hand three-process execution with Byzantine $\check{C}$ simulation C_0 through C_1. So validity forces them to decide 0. A similar argument using Byzantine $\check{A}$ shows the same for C_0.

opera is not a real proof, since we haven't actually shown that B can say exactly the right thing to keep A and C from guessing that B is evil.

Here is a real proof, which works by explicitly showing how to construct a bad execution for any given algorithm.[1] Consider an artificial execution where (non-Byzantine) A, B, and C are duplicated and then placed in a ring $A_0B_0C_0A_1B_1C_1$, where the digits indicate inputs. We'll still keep the same code for $n = 3$ on each process, but when A_0 tries to send a message to what it thinks of as just C we'll send it to C_1 while messages from B_0 will instead go to C_0. For any adjacent pair of processes (e.g. A_0 and B_0), the behavior of the rest of the ring could be simulated by a single Byzantine process ($\check{C}$), so each process in the 6-process ring behaves just as it does in some 3-process execution with 1 Byzantine process. It follows that all of the processes terminate and decide in the unholy 6-process Frankenexecution[2] the same value that they would in the corresponding 3-process Byzantine execution. So what do they decide?

Given two processes with the same input, say, A_0 and B_0, the giant execution is indistinguishable from an $A_0B_0\check{C}$ execution where $\check{C}$ is Byzantine (see Figure 10.1. Validity says A_0 and B_0 must both decide 0. Since this works for any pair of processes with the same input, we have each process deciding its input. But now consider the execution of $C_0A_1\check{B}$, where $\check{B}$ is Byzantine. In the big execution, we just proved that C_0 decides 0 and A_1 decides 1, but since the $C_0A_1\check{B}$ execution is indistinguishable from the big execution to C_0 and A_1, they do the same thing here and violate agreement.

[1]The presentation here is based on [AW04, §5.2.3]. The original impossibility result is due to Pease, Shostak, and Lamport [PSL80]. This particular proof is due to Fischer, Lynch, and Merritt [FLM86].

[2]Not a real word.

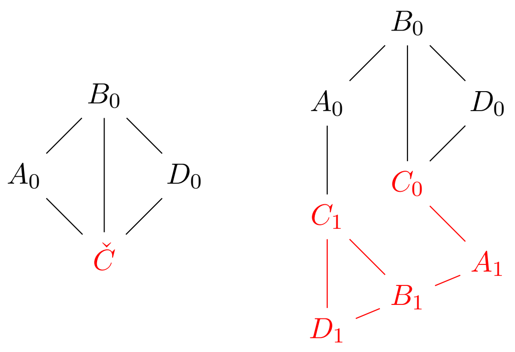

Figure 10.2: Four-process vs. eight-process execution in Byzantine agreement connectivity lower bound. Because Byzantine $\check{C}$ can simulate C_0, D_1, B_1, A_1, and C_1, good processes A_0, B_0 and D_0 must all decide 0 or risk violating validity.

This shows that with $n = 3$ and $f = 1$, we can't win. We can generalize this to $n = 3f$. Suppose that there were an algorithm that solved Byzantine agreement with $n = 3f$ processes. Group the processes into groups of size f, and let each of the $n = 3$ processes simulate one group, with everybody in the group getting the same input, which can only make things easier. Then we get a protocol for $n = 3$ and $f = 1$, an impossibility.

10.1.3 Minimum connectivity

So far, we've been assuming a complete communication graph. If the graph is not complete, we may not be able to tolerate as many failures. In particular, we need the connectivity of the graph (minimum number of nodes that must be removed to split it into two components) to be at least $2f + 1$. See [Lyn96, §6.5] for the full proof. The essential idea is that if we have an arbitrary graph with a vertex cut of size $k < 2f + 1$, we can simulate it on a 4-process graph where A is connected to B and C (but not D), B and C are connected to each other, and D is connected only to B and C. Here B and C each simulate half the processes in the size-k cut, A simulates all the processes on one side of the cut and D all the processes on the other side. We then construct an 8-process artificial execution with two non-faulty copies of each of A, B, C, and D and argue that if one of B or C can be Byzantine then the 8-process execution is indistinguishable to the remaining processes from a normal 4-process execution. (See Figure 10.1.)

An argument similar to the $n > 3f$ proof then shows we violate one of validity or agreement: if we replacing C_0, C_1, and all the nodes on one side of the $C_0 + C_1$ cut with a single Byzantine $\check{C}$, we force the remaining non-faulty nodes to decide their inputs or violate validity. But then doing the same thing with B_0 and B_1 yields an execution that violates agreement.

Conversely, if we have connectivity $2f+1$, then the processes can simulate a general graph by sending each other messages along $2f + 1$ predetermined vertex-disjoint paths and taking the majority value as the correct message. Since the f Byzantine processes can only corrupt one path each (assuming the non-faulty processes are careful about who they forward messages from), we get at least $f+1$ good copies overwhelming the f bad copies. This reduces the problem on a general graph with sufficiently high connectivity to the problem on a complete graph, allowing Byzantine agreement to be solved if the other lower bounds are met.

10.1.4 Weak Byzantine agreement

(Here we are following [Lyn96, §6.6]. The original result is due to Lamport [Lam83].)

Weak Byzantine agreement is like regular Byzantine agreement, but validity is only required to hold if there are no faulty processes at all. If there is a single faulty process, the non-faulty processes can output any value regardless of their inputs (as long as they agree on it). Sadly, this weakening doesn't improve things much: even weak Byzantine agreement can be solved only if $n \geq 3f + 1$.

Proof: As in the strong Byzantine agreement case, we'll construct a many-process Frankenexecution to figure out a strategy for a single Byzantine process in a 3-process execution. The difference is that now the number of processes in our synthetic execution is much larger, since we want to build an execution where at least some of our test subjects think they are in a non-Byzantine environment. The trick is to build a very big, highly-symmetric ring so that at least some of the processes are so far away from the few points of asymmetry that might clue them in to their odd condition that the protocol terminates before they notice.

Fix some protocol that allegedly solves weak Byzantine agreement, and let r be the number of rounds for the protocol. Construct a ring of $6r$ processes $A_{01}B_{01}C_{01}A_{02}B_{02}C_{02} \ldots A_{0r}B_{0r}C_{0r}A_{10}B_{10}C_{10} \ldots A_{1r}B_{1r}C_{1r}$, where each X_{ij} runs the code for process X in the 3-process protocol with input i. For each adjacent pair of processes, there is a 3-process Byzantine execution which is indistinguishable from the $6r$-process execution for that pair: since agreement holds in all Byzantine executions, each adjacent pair decides the same value in the big execution and so either everybody decides 0 or everybody decides 1 in the big execution.

Now we'll show that means that validity is violated in some no-failures 3-process execution. We'll extract this execution by looking at the execution of processes $A_{0,r/2}B_{0,r/2}C_{0,r/2}$. The argument is that up to round r, any input-0 process that is at least r steps in the ring away from the nearest 1-input process acts like the corresponding process in the all-0 no-failures 3-process execution. Since $A_{0,r/2}$ is $3r/2 > r$ hops away from A_{1r} and similarly for $C_{0,r/2}$, our 3 stooges all decide 0 by validity. But now repeat the same argument for $A_{1,r/2}B_{1,r/2}C_{1,r/2}$ and get 3 new stooges that all decide 1. This means that somewhere in between we have two adjacent processes where one decides 0 and one decides 1, violating agreement in the corresponding 3-process execution where the rest of the ring is replaced by a single Byzantine process. This concludes the proof.

This result is a little surprising: we might expect that weak Byzantine agreement could be solved by allowing a process to return a default value if it notices anything that might hint at a fault somewhere. But this would allow a Byzantine process to create disagreement revealing its bad behavior to just one other process in the very last round of an execution otherwise headed for agreement on the non-default value. The chosen victim decides the default value, but since it's the last round, nobody else finds out. Even if the algorithm is doing something more sophisticated, examining the $6r$-process execution will tell the Byzantine process exactly when and how to start acting badly.

10.2 Upper bounds

Here we describe two upper bounds for Byzantine agreement, one of which gets an optimal number of rounds at the cost of many large messages, and the other of which gets smaller messages at the cost of more rounds. (We are following §§5.2.4–5.2.5 of [AW04] in choosing these algorithms.) Neither of these algorithms is state-of-the-art, but they demonstrate some of the issues in solving Byzantine agreement without the sometimes-complicated optimizations needed to get all the parameters of the algorithm down simultaneously.

10.2.1 Exponential information gathering gets $n = 3f + 1$

The idea of **exponential information gathering** is that each process will do a lot of gossiping, but now its state is no longer just a flat set of inputs, but a tree describing who it heard what from. We build this tree out of pairs of the form $\langle \mathsf{path}, \mathsf{input} \rangle$ where path is a sequence of intermediaries with no repetitions and input is some input. A process i's state at each round is just a set of such pairs, represented by the variables $\mathsf{valpath}, i = \mathsf{input}$. At the end of $f + 1$ rounds of communication (necessary because of the lower bound for crash failures), each non-faulty process i attempts to untangle the complex web of hearsay and second-hand lies to compute the same decision value as the other processes, by computing reconstructed values $\mathsf{val}^*(\mathsf{path}, i)$ that, we hope, will eventually converge to the same values for all processes.

This technique was used by Pease, Shostak, and Lamport [PSL80] to show that their impossibility result is tight: there exists an algorithm for Byzantine agreement that runs in $f + 1$ synchronous rounds and guarantees agreement and validity as long as $n \geq 3f + 1$.

The algorithm is given in Algorithm 10.1. The communication phase is just gossiping, where each process starts with its only its input and forwards any values it hears about along with their provenance to all of the other processes. At the end of this phase, each process i has set $\mathsf{val}(\mathsf{path}, i)$ to some value value, where path spans all sequences of 0 to $f + 1$ distinct IDs and value is the input value forwarded along that path.

Because we can't trust these $\mathsf{val}(w, i)$ values to be an accurate description of any process's input if there is a Byzantine process in w, each process computes for itself reconstructed values $\mathsf{val}^*(w, i)$ that use majority voting to try to get a more trustworthy picture of the original inputs.

Formally, we think of the set of paths as a tree where w is the parent of wj for each path w and each ID j not in w. To apply EIG in the Byzantine model, ill-formed or missing messages from j are replaced by default values, but otherwise the data-collecting part of EIG proceeds as in the crash failure model. However, we compute the decision value from the last-round values recursively as follows. First, set $\mathsf{val}^*(w, i)$ for any path w with $|w| = f + 1$ to $\mathsf{val}(w, i)$. Then for each path w with $|w| < f + 1$, define $\mathsf{val}^*(w, i)$ to be the majority value among $\mathsf{val}^*(wj, i)$ for all j. Finally, have process i decide $\mathsf{val}^*(\langle\rangle, i)$. Note that this entire reconstruction process can be computed locally by each process, although we haven't yet shown that i's decision value $\mathsf{val}^*(\langle\rangle, i)$ will necessarily be the same as j's decision value $\mathsf{val}^*(\langle\rangle, j)$.

The majority rule for $w = \langle\rangle$ makes the decision value $\mathsf{val}^*(\langle\rangle, i)$ a majority of reconstructed inputs $\mathsf{val}^*(j, i)$. One way to think about this is that I never

```
// Set my value to my input
val(⟨⟩, $i$) ← input
for round ← $0 \dots f$ do
    // send step for this round
    for each non-repeating $w$, $|w| =$ round, $i \notin w$ do
        Send $\langle wi, \mathsf{val}(w, i)\rangle$ to all processes
    // receive step for this round
    for each non-repeating $w$, $|w| =$ round do
        if $j$ sent $\langle wj, v\rangle$ then
            // Record reported value
            $\mathsf{val}(wj, i) \leftarrow v$
        else
            // Record default value
            $\mathsf{val}(wj, i) \leftarrow 0$
// Compute decision value
for each path $w$ of length $f + 1$ with no repeats do
    $\mathsf{val}^*(w, i) \leftarrow \mathsf{val}(w, i)$
for $\ell \leftarrow f$ down to $0$ do
    for each non-repeating $w$, $|w| = \ell$ do
        $\mathsf{val}^*(w, i) \leftarrow \text{majority}_{j \notin w} \mathsf{val}^*(wj, i)$
Decide $\mathsf{val}^*(\langle\rangle, i)$
```

Algorithm 10.1: Exponential information gathering. Code for process i.

trust j to give me the correct value for wj—even when $w = \langle\rangle$ and j is claiming to report its own input—so instead I take a majority of values of wj that j allegedly reported to other people. But since I don't trust those other people either, I use the same process recursively to construct those reports, and hope that all the lies are eventually overcome by the truth.

10.2.1.1 Proof of correctness

This is just a sketch of the proof from [Lyn96, §6.3.2]; essentially the same argument appears in [AW04, §5.2.4].

We start with a basic observation that good processes send and record values correctly. Throughout the proof, we use $\mathsf{val}(w, i)$ for the final value of $\mathsf{val}(w, i)$ recorded by i.

Lemma 10.2.1. *If i and j are both non-faulty, then for all w, $\mathsf{val}(wj, i) = \mathsf{val}(w, j)$.*

Proof. Trivial: j sends $\langle wj, \mathsf{val}(w, i)\rangle$ to i, and i records it in $\mathsf{val}(wj, i)$. □

More involved is this lemma, which says that when we reconstruct a value for a trustworthy process at some level, we get the same value that it sent us. In particular this will be used to show that the reconstructed inputs $\mathsf{val}^*(j, i)$ are all equal to the real inputs for good processes.

Lemma 10.2.2. *If i and j are non-faulty, then for all w, $\mathsf{val}^*(wj, i) = \mathsf{val}(w, j)$.*

Proof. By induction on $f + 1 - |wj|$. If $|wj| = f + 1$, then $\mathsf{val}^*(wj, i) = \mathsf{val}(wj, i) = \mathsf{val}(w, j)$. If $|wj| < f+1$, then then $\mathsf{val}^*(wj, i) = \text{majority}_{k \notin wj} \mathsf{val}^*(wjk, i)$. The induction hypothesis says $\mathsf{val}^*(wjk, i) = \mathsf{val}(wj, k)$, which equals $\mathsf{val}(w, j)$ by Lemma 10.2.1. Now observe that there are at least $3f + 1 - |wj| \geq 2f + 1$ possible k, of which at most f are faulty, leaving a non-faulty majority all of which have $\mathsf{val}^*(wjk, i) = \mathsf{val}(w, j)$. □

We call a node w **common** if $\mathsf{val}^*(w, i) = \mathsf{val}^*(w, j)$ for all non-faulty i, j. Lemma 10.2.2 implies that wk is common if k is non-faulty. We can also show that any node whose children are all common is also common, whether or not the last process in its label is faulty.

Lemma 10.2.3. *Let wk be common for all k. Then w is common.*

Proof. Recall that, for $|w| < f + 1$, $\mathsf{val}^*(w, i)$ is the majority value among all $\mathsf{val}^*(wk, i)$. If all wk are common, then $\mathsf{val}^*(wk, i) = \mathsf{val}^*(wk, j)$ for all non-faulty i and j. so i and j compute the same majority values and get $\mathsf{val}^*(w, i) = \mathsf{val}^*(w, j)$. □

We can now prove the full result.

Theorem 10.2.4. *Exponential information gathering using $f + 1$ rounds in a synchronous Byzantine system with at most f faulty processes satisfies validity and agreement, provided $n \geq 3f + 1$.*

Proof. Termination: Protocol finishes after $f + 1$ rounds.

Validity: Immediate application of Lemmas 10.2.1 and 10.2.2 when $w = \langle\rangle$. We have $\mathsf{val}^*(j,i) = \mathsf{val}(j,i) = \mathsf{val}(\langle\rangle, j)$ for all non-faulty j and i, which means that a majority of the $\mathsf{val}^*(j,i)$ values equal the common input and thus so does $\mathsf{val}^*(\langle\rangle, i)$.

Agreement: Observe that every path has a common node on it, since a path travels through $f+1$ nodes and one of them is good. If we then suppose that the root is not common: by Lemma 10.2.3, it must have a not-common child, that node must have a not-common child, etc. But this constructs a path from the root to a leaf with no not-common nodes, which we just proved can't happen. $\square$

10.2.2 Phase king gets constant-size messages

The following algorithm, based on work of Berman, Garay, and Perry [BGP89], achieves Byzantine agreement in $2(f+1)$ rounds using constant-size messages, provided $n \geq 4f + 1$. The description here is drawn from [AW04, §5.2.5]. The original Berman-Garay-Perry paper gives somewhat better bounds, but the algorithm and its analysis are more complicated.

10.2.2.1 The algorithm

The main idea of the algorithm is that we avoid the recursive majority voting of EIG by running a vote in each of $f + 1$ *phases* through a **phase king**, some process chosen in advance to run the phase. Since the number of phases exceeds the number of faults, we eventually get a non-faulty phase king. The algorithm is structured so that one non-faulty phase king is enough to generate agreement and subsequent faulty phase kings can't undo the agreement.

Pseudocode appears in Algorithm 10.2. Each processes i maintains an array $\mathsf{pref}_i[j]$, where j ranges over all process IDs. There are also utility values $\mathsf{majority}$, $\mathsf{kingMajority}$ and $\mathsf{multiplicity}$ for each process that are used to keep track of what it hears from the other processes. Initially, $\mathsf{pref}_i[i]$ is just i's input and $\mathsf{pref}_i[j] = 0$ for $j \neq i$.

```
pref_i[i] = input
for j ≠ i do pref_i[j] = 0
for k ← 1 to f + 1 do
    // First round of phase k
    send pref_i[i] to all processes (including myself)
    pref_i[j] ← v_j, where v_j is the value received from process j
    majority ← majority value in pref_i
    multiplicity ← number of times majority appears in pref_i
    // Second round of phase k
    if i = k then
        // I am the phase king
        send majority to all processes
    if received m from phase king then
        kingMajority ← m
    else
        kingMajority ← 0
    if multiplicity > n/2 + f then
        pref_i[i] = majority
    else
        pref_i[i] = kingMajority
return pref_i[i]
```

Algorithm 10.2: Byzantine agreement: phase king

The idea of the algorithm is that in each phase, everybody announces their current preference (initially the inputs). If the majority of these preferences is large enough (e.g., all inputs are the same), everybody adopts the majority preference. Otherwise everybody adopts the preference of the phase king. The majority rule means that once the processes agree, they continue to agree despite bad phase kings. The phase king rule allows a good phase king to end disagreement. By choosing a different king in each phase, after $f + 1$ phases, some king must be good. This intuitive description is justified below.

10.2.2.2 Proof of correctness

Termination is immediate from the algorithm.

For validity, suppose all inputs are v. We'll show that all non-faulty i have $\mathsf{pref}_i[i] = v$ after every phase. In the first round of each phase, process i receives at least $n - f$ messages containing v; since $n \geq 4f + 1$, we have $n - f \geq 3f + 1$ and $n/2 + f \leq (4f + 1)/2 + f = 3f + 1/2$, and thus these $n - f$ messages exceed the $n/2 + f$ threshold for adopting them as the new preference. So all non-faulty processes ignore the phase king and stick with v, eventually deciding v after round $2(f + 1)$.

For agreement, we'll ignore all phases up to the first phase with a non-faulty phase king. Let k be the first such phase, and assume that the pref values are set arbitrarily at the start of this phase. We want to argue that at the end of the phase, all non-faulty processes have the same preference. There are two ways that a process can set its new preference in the second round of the phase:

1. The process i observes a majority of more than $n/2 + f$ identical values v and ignores the phase king. Of these values, more than $n/2$ of them were sent by non-faulty processes. So the phase king also receives these values (even if the faulty processes change their stories) and chooses v as its majority value. Similarly, if any other process j observes a majority of $n/2 + f$ identical values, the two $> n/2$ non-faulty parts of the majorities overlap, and so j also chooses v.

2. The process i takes its value from the phase king. We've already shown that i then agrees with any j that sees a big majority; but since the phase king is non-faulty, process i will agree with any process j that also takes its new preference from the phase king.

This shows that after any phase with a non-faulty king, all processes agree. The proof that the non-faulty processes continue to agree is the same as for validity.

10.2.2.3 Performance of phase king

It's not hard to see that this algorithm sends exactly $(f+1)(n^2+n)$ messages of 1 bit each (assuming 1-bit inputs). The cost is doubling the minimum number of rounds and reducing the tolerance for Byzantine processes. As mentioned earlier, a variant of phase-king with 3-round phases gets optimal fault-tolerance with $3(f+1)$ rounds (but 2-bit messages). Still better is a rather complicated descendant of the EIG algorithm due to Garay and Moses [GM98], which gets $f+1$ rounds with $n \geq 3f+1$ while still having polynomial message traffic.

Chapter 11

Impossibility of asynchronous agreement

Last updated 2025. Some material may be out of date.

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

[1] Unless you augment the basic model in some way, say by adding randomization (Chapter 24) or failure detectors (Chapter 13).

surprisingly limited, and so most of the proof is the same for both shared memory (even strong versions of shared memory that support operations 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 28.

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.

With asynchronous scheduling and required termination, this is equivalent to a limited version of fairness in which one process is labeled as faulty and the adversary is not required to deliver messages from that process. Having an active failure step (as opposed to the adversary just choosing internally not to deliver some process's messages) mostly just lets us more easily describe which process the adversary is doing this to.

[2]Chapter 20.

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 $e_1 e_2 \dots e_k$ 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

The FLP results extends to any asynchronous model where it is impossible to tell which of two events happened first. The main idea is to replace the definition of a step to whatever is available in the new model, and adapt the resulting case analysis of 0-valent $De'e$ vs 1-valent Dee' as appropriate. 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 (two reads) or only one process can tell whether both happened (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.

Loui and Abu-Amara [LAA87] first proved this generalization to shared memory using standard read-write registers. Herlihy [Her91b] later provided similar arguments for a wide variety of shared-memory primitives that may provide additional operations beyond reads and writes. We will see many of these latter arguments in Chapter 19.