Lecture 3

Impossibility of Consensus

In the previous lecture, we saw that it is possible to simulate synchronous algorithms in asynchronous systems. Today, we will see that a basic fault-tolerance task, *consensus*, is unsolvable in asynchronous systems. In the exercises, we will see that consensus is straightforward in synchronous systems, separating synchronous and asynchronous systems beyond differences in efficiency.

3.1 The Problem

A standard formulation of the (binary) consensus problem is given as follows. Each of n nodes is given a binary input b_i, $i \in \{1, \ldots, n\}$. Nodes may *crash* during the execution. A node that crashes is *faulty*, while nodes that do not crash are *correct*. Correct nodes $i \in [n]$ are to compute an output o_i such that the following properties hold.

Agreement Correct nodes i output the same value $o = o_i$.

Validity If all nodes have the same input b, then $o = b$.

Termination All correct nodes decide on an output and terminate.

Being able to solve this problem can, e.g., be useful for control of a plane. For safety reasons, there are several computers in case some of them fail. Suppose they need to decide between two possible courses for the plane, at least one of which is safe. If the computers each compute an opinion b_i based on the data they have, you surely want the decision to satisfy all three properties:

Validity If the data clearly prefers one route over the other, this decision should be taken! Otherwise: plane crash.

Agreement *Some* decision must be taken even if the data is inconclusive (the computers compute different values b_i). The plane must take one of the two routes! Otherwise: plane crash.

Termination This decision must be taken at some point. In fact, probably soon, which is why the time complexity of consensus algorithms is important, too! Otherwise: plane crash.

Figure 3.1: This is not supposed to happen!

Note that this problem is a no-brainer in absence of faults. Just pick a leader (e.g., the node with smallest identifier) and decide on its input! But what if this node crashes? In a synchronous system someone will notice, but in an asynchronous system there is no way to be sure that it's not just a bad case of excruciatingly slow message delivery...

We need to specify the model in which we want to consider the problem. We will use a model that is stronger than the message passing model (we will see later why), the asynchronous *shared memory* model. Here, there is some common memory accessible by all n nodes that is used to communicate. Nodes read and write *registers* of this memory *atomically*. This means that nodes read the entire register in one go or (over)write the content of a register without anyone else interfering. For convenience, we assume that all registers are initialized with a special symbol $\perp$. The catch now is that a scheduler decides who's next – and since we're talking asynchrony here, it is under no obligation regarding which other nodes it schedules (and how often) before it picks a specific node that wants to read or write. However, it is required to schedule non-crashed nodes *eventually*. Any node that intends to read or write is scheduled (or crashes) after finitely many steps. This property is called *fairness*. The scheduler may also decide to *crash* a node, simply meaning that it will not be scheduled again.

As usual, nodes have unique identifiers, initially know their input value only, and local computations are "free." It's convenient to assume that a node performs all its initial local computations and those after a read/write instantaneously. Thus, a node is always either waiting to perform a read or write operation, is crashed, or is terminated; local termination occurs when a node decides at the end of a step that it's done and outputs a value.

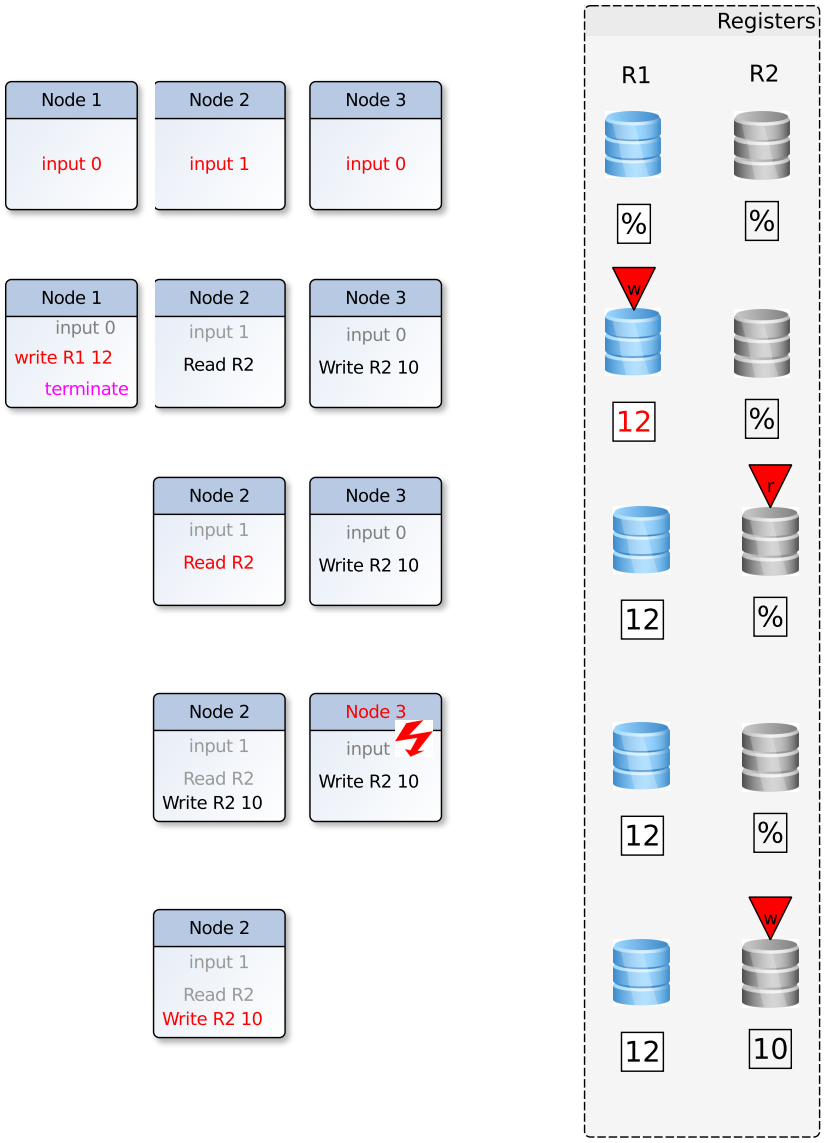

Figure 3.2: Sample execution of shared memory system with 3 nodes and 2 shared registers. The depicted execution is $(b_1 = 0, b_2 = 1, b_3 = 0, \text{write}_1(R1, 12), \text{term}_1, \text{read}_2(R2), \text{crash}_3)$. The currently executed operation is marked red, gray operations are already executed and black operations are currently outstanding.

Remarks:

- Dropping any of the requirements of agreement, validity, or termination renders the problem trivial. Think of the respective "solutions!"
- Observe that fairness basically means that it's not ok to crash a node without saying so. This is relevant because a crashed node does not have to decide or, if it already decided, have the same output as others.
- With fairness, one can define asynchronous rounds like for message passing: within one time unit, each node is guaranteed to be scheduled at least once.
- We assume the powerful shared memory communication and benign faults (there's much worse than clean crashes out there, but that's a tale for another day!). This makes the impossibility we will show a strong result.
- On the other hand, we consider asynchronous communication and deterministic algorithms, so do not despair!

3.2 Getting Started

Today's main result was surprising and a big deal when it was shown first. It was surprising both because it's not easy to show and because quite a few people believed that asynchronous consensus *is* possible. It will be much easier for us, and that's because the right definitions will point us in the right direction.[1]

Definition 3.1 (Executions)**.** *An execution of an algorithm is given by a sequence of read and write operations, crashes, and terminations, alongside the initial inputs given to the nodes; naturally, the decision whether a node terminates, reads, or writes (and if so what) in its next step is made by the algorithm.*

Note that, since we require that all nodes that do not crash must terminate, all executions that are relevant to us are of finite length. Also, as stated earlier, we will consider fair executions only.

We now can state the main result.

Theorem 3.2 (FLP (Fischer, Lynch & Patterson))**.** *There is no algorithm that solves the consensus problem in all fair executions with at most one fault.*

As mentioned, good definitions are pivotal. We will need two key concepts. The first is called *indistinguishability*. Note that while Definition 3.1 is about the entire network, the following definition is about how an execution looks like at a specific node:

Definition 3.3 (Indistinguishable Executions)**.** *Two executions are* indistinguishable *at node i, iff in both executions i has the same input, performs the same sequence of read and write operations, and all the read operations return the same values in both executions.*

[1] The professor of one of my math courses once said that *definitions* are even more important than theorems, because the right definition tells us how to look at things and paves the way for the big results.

If two executions are indistinguishable at node i, it must behave the same way in both executions.

Lemma 3.4. *If two executions are indistinguishable at node i, the write operations of i in both executions are identical. If it terminated, the output values are identical. If it hasn't terminated yet, its next action is the same in both executions.*

Proof. By induction, the memory state of and values written by i are the same in the respective steps of each execution. Hence i's output value or next step, respectively, is also the same. □

The second definition looks even simpler.

Definition 3.5 (Bivalency and Univalency)**.** *For $b \in \{0, 1\}$, an execution of a consensus algorithm is b-*valent*, if any possible continuation of the execution results in output b. It is* univalent*, if it is b-valent for some b. Otherwise, it is* bivalent*.*

Combining these two notions, we obtain a crucial observation that will be at the heart of our reasoning.

Corollary 3.6. *If two executions are indistinguishable at all non-crashed nodes and each shared register contains the same value at their end, they have the same valency (i.e., both are 0-, both are 1-, or both are bivalent).*

Proof. By (inductive use of) Lemma 3.4, any extension of one execution is also a valid extension of the other, and the result will be two indistinguishable executions: every read operation will return the same value in both executions. Thus, outputs in such a pair of executions must be identical. Now the claim readily follows from the definition of bi- and univalency. □

Here's the plan:

1. Show that there are bivalent executions or validity is violated.
   (a) If validity holds, use it to show that there are 0- and 1-valent executions.
   (b) Infer that there must be a configuration for which one node's input makes the difference.
   (c) Conclude that crashing/not crashing the node must result in different outputs in some execution.
2. For any node i, show that we can extend any bivalent execution to another bivalent execution such that i takes another step; alternatively, there is an execution violating agreement.
   (a) For a bivalent execution that has no bivalent extension with another step of i, there are 0- and 1-valent extensions involving another step of i.
   (b) Infer that there must be a configuration for which swapping the steps of nodes i and some $j \neq i$ makes the difference between 0- and 1-valency.

(c) Perform a case analysis proving that agreement is violated in some execution (using Lemma 3.4, Corollary 3.6, and the 0-/1-valency of the extensions).

3. Conclude that if agreement and validity hold, an infinite fair execution exists (i.e., termination does not hold).

As you can see, many of the above statements require that some of the properties of a consensus algorithm hold. For simplicity, we will assume that we have an algorithm solving consensus and ultimately derive a contradiction. Apart from this, the structure of the proof remains exactly as outlined above.

3.3 Step 1: Bivalent Executions Exist

In the following, let $\mathcal{A}$ be a consensus algorithm, i.e., one that satisfies agreement, validity, and termination. First, we use validity to show that there must be at least one bivalent execution.

Lemma 3.7. *$\mathcal{A}$ has a bivalent execution without crashes.*

Proof. For $j \in [n+1]$, consider the execution $\mathcal{E}_j$ that's simply given by the inputs $b_i = 0$ for all $i > j$ and $b_i = 1$ for $i \leq j$ (i.e., nothing has happened yet except for the inputs being specified). If any of these executions is bivalent, we're done, so let's suppose for contradiction that they are all univalent.

If $j = 0$, $b_i = 0$ for all $i \in \{1, \ldots, n\}$, and validity implies that the output is 0. Likewise, the execution with $j = n$ is 1-valent. Hence, there must be some $j \in [n]$ such that $\mathcal{E}_j$ is 0-valent and $\mathcal{E}_{j+1}$ is 1-valent. Since nothing has happened yet, both executions are indistinguishable to all nodes but j, which has different input in both executions. Thus, crashing j yields two executions of different valency that are indistinguishable at all non-crashed nodes, where the shared registers haven't been touched yet. This contradicts Corollary 3.6! □

Remarks:

- We used the *possibility* of a fault to show that there is a bivalent execution. However, we didn't "use up" the fault, we have a fault-free bivalent execution!

3.4 Step 2: Extending Bivalent Executions

Next, we show that, given a bivalent execution and a node i, a "follow-up" execution exists that is also bivalent and in which i performs a step. This last bit is crucial, because it ensures that a bivalent execution can be "kept bivalent" *even in a fair schedule.*

We start with a helper lemma ensuring that we can extend a bivalent execution to force either decision without crashing a node.

Lemma 3.8. *Given a bivalent execution $\mathcal{E}$ of $\mathcal{A}$ and $b \in \{0, 1\}$, we can extend $\mathcal{E}$ to a b-valent execution $\mathcal{E}'$ without any further crashes.*

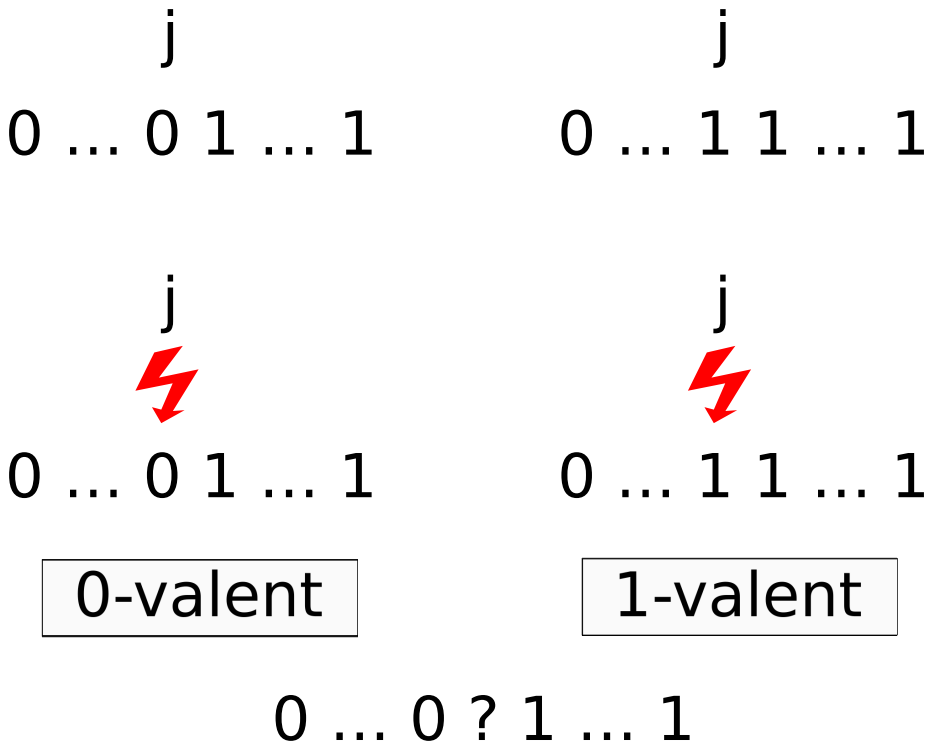

Figure 3.3: Key argument of Lemma 3.7. Assuming that no bivalent execution exists, validity implies that we can find a pair of "executions" (i.e., inputs) for which only the input of a single node differs, but one execution is 1- and the other 0-valent. Crashing this node, which is the only one knowing about the difference, right away, yields a contradiction.

Proof. By definition of bivalency, there is *some* execution $\mathcal{E}_b$ extending $\mathcal{E}$ that is b-valent. However, it might contain crashes. We extend $\mathcal{E}_b$ further to $\mathcal{E}'_b$, in which some node decides on b and terminates; this is feasible by termination of $\mathcal{A}$. Now we remove all crashes from $\mathcal{E}'_b$, resulting in execution $\mathcal{E}'$. By Lemma 3.4 and the fact that the crashed nodes do not change the contents of registers in either execution, the node still decides on b and terminates. Thus, $\mathcal{E}'$ must be b-valent by agreement, and by construction it contains no further crashes. □

Now we can proceed to extending (fault-free) bivalent executions in a way keeping them bivalent (and fault-free).

Lemma 3.9. *Given a bivalent execution $\mathcal{E}$ of $\mathcal{A}$ and a non-crashed node $i \in [n]$, we can construct a bivalent execution with an additional step of i. If $\mathcal{E}$ is fault-free, so is the new execution.*

Proof. Refer to Figure 3.4. Clearly, i cannot be terminated in $\mathcal{E}$, as otherwise the execution must be univalent by agreement. Let i take an additional step. If the extended execution $\mathcal{E}_0$ is still bivalent, we're done. Otherwise, assume w.l.o.g. that $\mathcal{E}_0$ is 0-valent. Because $\mathcal{E}$ is bivalent, by Lemma 3.8 there is also

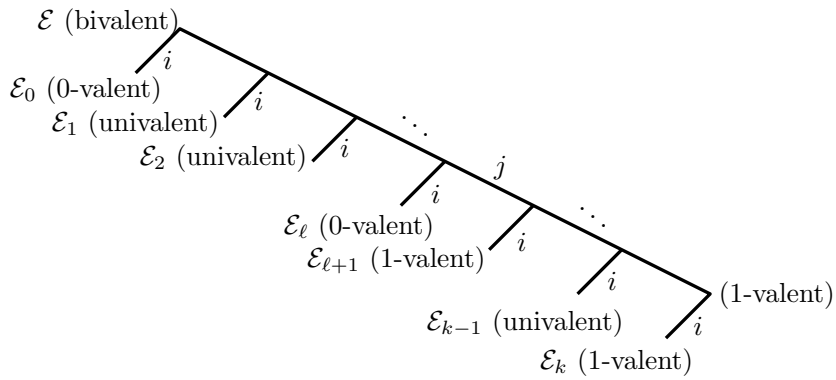

Figure 3.4: By assumption there is no bivalent extension of $\mathcal{E}$ that contains an additional step of node i. The switch from 0- to 1-valency happens between $\mathcal{E}_\ell$ and $\mathcal{E}_{\ell+1}$.

an extension of $\mathcal{E}$ that is 1-valent and contains no further crashes. Take such an extension, denote by k the number of additional steps, and let $\mathcal{E}_k$ be this extension plus an additional step of i.[2] Note that $\mathcal{E}_k$ is 1-valent, as any extension of a 1-valent execution is 1-valent. In summary, we have two extensions of $\mathcal{E}$, both with a step of i at the end, and either 0 ($\mathcal{E}_0$) or $k \neq 0$ ($\mathcal{E}_k$) intermediate steps. $\mathcal{E}_0$ is 0-valent and $\mathcal{E}_k$ is 1-valent.

Next consider the executions $\mathcal{E}_\ell$, $\ell \in [k+1]$, which are $\mathcal{E}$ followed by the next ℓ steps that happen in $\mathcal{E}_k$, and each of them with a final step of i (Figure 3.4). If any of these are bivalent, we're done. Otherwise, as we know that $\mathcal{E}_0$ is 0-valent and $\mathcal{E}_k$ is 1-valent, there must be some $\ell \in [k]$ so that $\mathcal{E}_\ell$ is 0-valent and $\mathcal{E}_{\ell+1}$ is 1-valent.

Suppose that j is the node that takes the final step before node i in execution $\mathcal{E}_{\ell+1}$. Note that both executions are indistinguishable at all nodes but i and j, and any difference must come from the final one or two steps. To complete the proof, we go through all possible cases and lead each of them to a contradiction with Corollary 3.6.

$i = j$: In this case, letting i take another step in $\mathcal{E}_\ell$ results in $\mathcal{E}_{\ell+1}$. But one is 0- and the other is 1-valent. Contradiction!

j does not write: We crash j at the end of both $\mathcal{E}_\ell$ and $\mathcal{E}_{\ell+1}$. The executions are indistinguishable at all nodes but the crashed j, and i did the same write (or read) in both executions, resulting in identical content of the shared registers. However, one execution is 0- and the other 1-valent. Contradiction!

i does not write: We let j take another step in $\mathcal{E}_\ell$, which is the same as the one in $\mathcal{E}_{\ell+1}$; since i didn't write, it is the only node that can distinguish the two executions, and again the shared registers have identical contents in both executions. Crashing i thus yields a contradiction.

[2] Again, as soon as i terminates, the execution must become univalent, so either i can still take a step or i just terminated in the last step; in the latter case, we just use the execution directly without adding a step of i.

i and j write to different registers: We let j take its step in $\mathcal{E}_\ell$ and obtain an execution that is indistinguishable at all nodes. Since the two writes do not interfere, the shared registers' content is identical, too. One execution is 0-, the other 1-valent. Contradiction!

i and j write to the same register: As i overwrites j's write in $\mathcal{E}_{\ell+1}$, j is the only node that can distinguish $\mathcal{E}_\ell$ and $\mathcal{E}_{\ell+1}$, and the register contains the value written by i at the end of both executions. Crashing j yields a contradiction!

Since all possibilities lead to a contradiction, we must have had the situation that we encountered a bivalent execution earlier on. Also, all the executions $\mathcal{E}_\ell$, $\ell \in [k+1]$, contained at least one more step of i than $\mathcal{E}$. □

Remarks:

- This proof critically relies on the assumption that only a *single* register can be written atomically. What happens if it's possible to write concurrently to several?

3.5 Step 3: Reaching Contradiction

All that remains is to wrap things up.

Proof of Theorem 3.2. Assume for contradiction that a consensus algorithm $\mathcal{A}$ exists that tolerates a single fault, i.e., in all fair executions with at most one crash agreement, validity, and termination hold. By Lemma 3.7, there is a bivalent execution of $\mathcal{A}$ without crashes. By Lemma 3.9, we can extend any such execution to a bivalent execution without crashes that includes an additional step of an arbitrary node $i \in [n]$. We apply the lemma inductively in a round-robin fashion; in the kth step of the induction, we add a step of node $k \mod n$. The result is an infinite, fair, bivalent execution without crashes. This contradicts the condition that the algorithm must terminate in *all* fair executions with at most one crash! □

3.6 How about Message Passing?

Fine, we can't do it in this shared memory setting. But is consensus possible in the asynchronous message passing model? At least for *some* graphs? To answer this question, we need to specify what it means that a node crashes in the message passing model.

Definition 3.10 (Crash Faults in the Message Passing Model)**.** *A node may crash at any point in the execution, after which it does not respond to any further events. It may also crash when responding to an event. In this case, it sends an arbitrary subset of the messages it would send if it did not crash.*

This definition takes into account that it's virtually impossible to make sure that a crashing node sends either everything or nothing – that would be very similar to writing *multiple* registers atomically! Each individual message *is*

sent and received "atomically," which is justified since any message that is not transmitted and received completely can simply be dropped.

It may not seem like it, but basically we already have the answer. We use a simulation argument!

Lemma 3.11 (Simulation of Message Passing)**.** *If, for any simple graph $G = (V, E)$, an asynchronous message passing algorithm solving consensus with at most one crash fault on G exists, then there is an asynchronous shared memory algorithm on $|V|$ nodes that solves consensus in all fair executions with at most one fault.*

Proof. We "translate" the message passing system to a shared memory system. We use the same set of nodes. For each edge $e = \{v, w\}$, we add registers $R_{v,w,i}$ and $R_{w,v,i}$, $i \in \mathbb{N}$, initialized to $\perp$ (meaning not used). For each neighbor w, v maintains two local counters $s_{v,w}$ and $r_{v,w}$, the number of sent and received messages for this node, respectively (initially 0). We simulate the message passing algorithm as follows. Initially, each node v performs its local computations and decides on the messages to send. Then, for each neighbor w to which it sends a message, it increases $s_{v,w}$ and writes the content of the message to $R_{v,w,s_{v,w}}$. Once this is complete, it executes a *busy-wait*. Cycling through its neighbors, w, it keeps reading $R_{v,w,r_{v,w}+1}$ until $R_{v,w,r_{v,w}+1} \neq \perp$ for some w. When this happens, it executes the code of the asynchronous algorithm for reception of a message with the content equal to that of $R_{v,w,r_{v,w}+1}$ from w and increases $r_{v,w}$. Resulting messages are resolved as above and the busy-wait recommences (unless the node terminates, of course).

It's straightforward to see that each "sent message" is eventually "received" (unless the receiving node terminates or crashes before this happens, which is ok), and since the shared memory algorithm does the same computations and "sends" the same messages, it will produce the same outputs as some corresponding execution of the message passing algorithm. Thus, agreement, validity, and termination of the shared memory version are inherited from the original algorithm. □

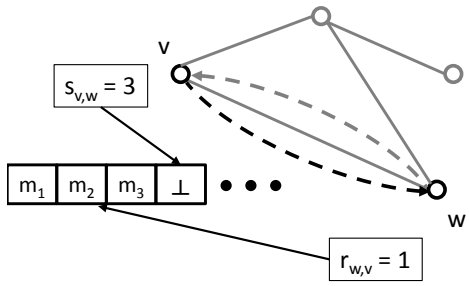

Figure 3.5: Construction for simulation of a message passing algorithm in shared memory. Depicted are only the registers for the edge $\{v, w\}$ for the direction from v to w. Node v will write each message to a new register using its local counter $s_{v,w}$. Node w will increase its counter $r_{v,w}$ whenever it reads a value that is not $\perp$, meaning it "received" the next message from v.

This lemma extends the previous impossibility to the asynchronous message passing model.

Corollary 3.12. *There is no algorithm that solves the consensus problem in the asynchronous message passing model with at most one crash fault.*

Proof. If such an algorithm existed, by Lemma 3.11 there would also be an algorithm solving consensus in all fair executions of the asynchronous shared memory model with at most one crash fault. By Theorem 3.2, such an algorithm does not exist. □

Remarks:

- We're doing something that might seem weird here. In the simulation, we use infinitely many shared registers (as there can be an unbounded number of messages under way in the message passing system), and these registers have infinite size (as messages may be arbitrarily large). However, we're talking about an impossibility result here: *Even* with such an impossible-to-build system, we *still* couldn't solve the problem!
- Note also that the simulation will actually ensure FIFO (first-in-first-out) order of message reception. Again, this makes the impossibility result only stronger. Also if message delivery is guaranteed to happen in FIFO order, the problem cannot be solved!
- Originally, the FLP theorem was shown for the message passing model. Showing it for shared memory and then using a simulation argument as done here is much simpler, yet we get the result for the more powerful shared memory model along the way!

What to take Home

- Knowing that certain things *cannot* be done is really important, as it keeps us from trying to do these things.
- Actually, it will not really keep us from trying, as it's important to solve these problems. However, such results show where one can change the model (i.e., add some helpful, hopefully realizable assumptions), so that they become solvable.
- Finding the right definitions can be the most important part of the job.
- Simulation arguments are also very powerful tools for lower bounds. FLP is a great example for this, as it's much easier to prove the result for shared memory and transfer it to message passing than taking the message passing model head on!

Bibliographic notes

Fischer, Lynch, and Patterson showed the original theorem about message passing systems, in a model slightly, but insubstantially different from the asynchronous message passing model given in Lecture 2 [FLP85]. Loui and Abu-Amara [LAA87] extended the result to the shared memory setting; strictly

speaking, Theorem 3.2 is to be attributed to them, but Fischer, Lynch, and Patterson developed the underlying technique. Later it was discovered that the impossibility of consensus and generalizations can be shown using topological tools [HS99].

Bibliography

[FLP85] Michael J. Fischer, Nancy A. Lynch, and Mike Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32(2):374–382, 1985.

[HS99] Maurice Herlihy and Nir Shavit. The Topological Structure of Asynchronous Computability. *J. ACM*, 46(6):858–923, 1999.

[LAA87] Michael C Loui and Hosame H Abu-Amara. Memory requirements for agreement among unreliable asynchronous processes. *Advances in Computing Research*, 4(163–183), 1987.