Foundations of Blockchains
Lectures #4 & 5: The Asynchronous Model and the FLP Impossibility Theorem (ROUGH DRAFT)*

Tim Roughgarden†

1 The Upshot

1. The asynchronous model of communication is the polar opposite of the synchronous model: no shared global clock, and no guarantees on message delivery (other than eventual delivery).
2. Designing fault-tolerant protocols with provable guarantees is challenging in the asynchronous model because of the de facto conspiracy between Byzantine nodes and the sequence of message deliveries.
3. Because the asynchronous model makes only minimal assumptions about the communication network, any consensus protocol with provable guarantees in it is automatically interesting.
4. The Byzantine agreement (BA) problem is a variant of the Byzantine broadcast (BB) problem in which there is no distinguished sender node and every node has a private input.
5. A consensus protocol solves the BA problem if it satisfies termination (defined as for BB), agreement (defined as for BB), and validity (if all honest nodes have the same private input, that should also be their common output).
6. The FLP (Fischer-Lynch-Paterson) impossibility theorem states that no deterministic protocol solves the BA problem in the asynchronous model, even with $f = 1$ and the most benign type of faulty node (a "crash fault").

*©2021–2022, Tim Roughgarden.
†Email: tim.roughgarden@gmail.com. The preparation of these notes was partially supported by a seed grant from the Columbia-IBM Center for Blockchain and Data Transparency. Thanks to Ittai Abraham for helpful comments on an earlier draft of these notes.

7. The same impossibility result holds for the state machine replication problem.

8. The high-level proof plan for the FLP impossibility theorem is to exhibit, for any protocol guaranteed to satisfy agreement and validity on termination, an infinitely long protocol trajectory (ruling out the termination property).

9. The first step of the proof shows that, for every such protocol, there exists a choice of nodes' private inputs such that the protocol begins in an ambiguous state (with the adversary in control over the honest nodes' eventual outputs).

10. Such a choice exists because the protocol is assumed to satisfy validity and because a Byzantine node could control the perceived value of a pivotal private input.

11. The second step of the proof shows how to extend a sequence of ambiguous configurations: given an ambiguous configuration and a message (r, m) in its pool, it's possible to eventually deliver (r, m) without resolving the ambiguity.

12. Starting from an ambiguous initial configuration and applying the second step over and over (with (r, m) chosen as the oldest message remaining in the pool) produces an arbitrarily long sequence of ambiguous configuration in which every message is eventually delivered.

13. The proof of the second step uses breadth-first search from an ambiguous configuration to identify the nearest candidate configuration at which the delivery of the chosen message (r, m) might preserve ambiguity.

14. This candidate configuration does, in fact, satisfy this property because a Byzantine node could control the perceived delivery order of a pivotal pair of messages.

15. The FLP impossibility result demonstrates that consensus in the asynchronous model is fundamentally harder than in the synchronous model, and it guides us to the "sweet spot" partially synchronous model that is the topic of Lecture 6.

2 Relaxing the Synchronous Assumption

These lectures introduce the asynchronous model of communication and prove what's possibly the most famous impossibility result in distributed computing, the FLP impossibility theorem. This will probably be the longest and trickiest proof that we do in the entire lecture series. Understanding it will be a feather in your cap and put you in quite rarified company.[1]

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

2.1 Recap and Context

Lecture 2 presented the Dolev-Strong protocol as a solution to the Byzantine broadcast problem. That protocol satisfied termination, agreement, and validity, no matter how many nodes were Byzantine (i.e., could deviate arbitrarily from the protocol). Using the Dolev-Strong protocol as a subroutine, we constructed an equally fault-tolerant protocol for the (multi-shot) state machine replication (SMR) problem satisfying consistency and liveness.

We proved these guarantees for the Dolev-Strong protocol (and hence our SMR protocol) under three assumptions. First, that the set of nodes and their names are known a priori (the "permissioned setting"). We will continue to make this assumption in these lectures (waiting until Lecture 9 to relax it, in the context of sybil-resistance and proof-of-work).

Second, the "public key infrastructure (PKI)" assumption: secure digital signature schemes exist, every node has a distinct public key-private key pair, and all nodes' public keys are common knowledge at the start of the protocol. As we saw in Lecture 3, the PKI assumption fundamentally changes whether or not the Byzantine broadcast problem is solvable in the synchronous model (with $f \geq n/3$, where n is the number of nodes and f is the maximum number of Byzantine nodes). In the asynchronous model studied in these lectures, it turns out not to matter (the FLP impossibility theorem holds even with the PKI assumption).

The third assumption (really, two subassumptions) is the one we're keen to relax in this lecture: (i) all nodes share a global clock; (ii) every message sent during a time step t arrives at its destination prior to the start of time step $t + 1$. We might be able to live with (i), but (ii) is completely unreasonable for a protocol operating over the Internet. For example, if we define time steps (rather generously) to be two seconds long, then under "normal operating conditions" we might expect subassumption (ii) to hold. But we all have first-hand experience with Internet delays longer than two seconds. There are a million reasons why this could happen. Maybe the Internet (or at least your neighborhood in it) is congested, with lots of packets getting dropped and retransmitted. Maybe the BGP routing protocol is having a really bad day and sending packets along highly suboptimal routes. Maybe there are some serious network outages. Or maybe the delays are being caused deliberately by a malicious actor— a "denial-of-service (DoS)" attack.[2]

2.2 Time Inflation: Failed Attempts at Relaxing the Model

Maybe the simplest way to relax the problematic subassumption (ii) in the synchronous model is allow a message to be delayed up to some number of time steps—for example, perhaps every message arrives after somewhere between 1 and 100 time steps. Unfortunately, this idea does not really generalize the synchronous model at all. Think about the Dolev-Strong protocol, for instance. In that protocol, the sender sent out messages at time step 0, and the non-senders exchanged cross-checking messages in time step 1, time step 2, and so on. If we're told that messages might get delayed up to 100 time steps, the obvious response is to have the non-senders instead exchange messages only at time step 100, time step 200,

[2]DoS attacks are not hard to pull off in practice. E.g., to cut a target machine off from the rest of the network, you could rent a botnet to pummel the machine with an overwhelming number of data packets.

and so on. The behavior and guarantees of the modified protocol then match those of the original protocol in the original model.[3]

This attempt at relaxing the synchronous model is deeply unsatisfying for two semi-related reasons. First, it did not force us to confront the main issue, that a protocol operating over the Internet necessarily must deal with unexpected outages and attacks. Second, it naturally led us to stupid protocols that spend most of their time idle. For example, in the time-inflated version of the Dolev-Strong protocol, even if all the sender's messages arrive at the non-senders by time step 1, the non-senders will wait until time step 100 before doing anything, just to make sure that everybody has the chance to receive all the messages destined to them.

A practical protocol can't afford to have its speed dictated by the worst-possible message delay, and should be fast whenever the underlying communication network is fast. This is the basic intuition behind (one version of) the partially synchronous model, discussed in Lecture 6. But first, let's explore what's possible with only the most minimal assumptions about the reliability of message delivery.

3 The Asynchronous Model

Informal description. The asynchronous model of communication represents the polar opposite of the synchronous model, with the two subassumptions replaced by non-assumptions. First, there will be no shared global clock, and thus no (even approximately) shared notion of time.[4] Second, messages may suffer arbitrarily long (finite) delays—a protocol designed for the asynchronous model must be ready for anything.

The asynchronous model does make a minimal assumption about message delivery: every message is eventually delivered to the intended recipient. (Without this assumption, it's possible that no messages ever arrive and trivially consensus is impossible to reach. We want to show that consensus is impossible *even if* all messages eventually arrive.) There is no a priori bound on how long delivery takes for any given message, however—it may well arrive after the delivery of one billion other messages that were sent after it.

Formal description. To rigorously prove an impossibility result, we need a completely formal mathematical model. Here it is:

The Asynchronous Model

- M denotes the pool of outstanding (not-yet-delivered) messages (initial-

[3]Similarly, you could imagine relaxing subassumption (i) by allowing nodes' clocks to drift away from each other, but only up to a bounded amount that is known up front (say, 10 minutes). The same "time inflation" idea shows that this modification to the model also does not take you outside of the synchronous model. (Working that statement out carefully is a good exercise.)

[4]A node is free to use its own local notion of time, but its estimate of the passage of time may be completely unrelated to those of the other nodes.

ized to $\{(r, \perp)\}_{r=1}^{n}$)

- while(TRUE):
  - an arbitrary message $(r, m) \in M$ is delivered (to recipient r);

    (subject to the eventual delivery constraint)
  - r can add any number of messages to M.

It might be helpful to think of the iterations of the main while loop as time steps, but keep in mind that, in the asynchronous model, nodes have no idea how many iterations have been executed. A *message* (r, m) sent by a node specifies two things, the recipient r (one of the n nodes) and the payload m (which is arbitrary content). The model is purely event-driven, with messages sent by a node only in response to receiving a new message. Every sent message (r, m) will be delivered eventually (to r), but the order in which messages are delivered is arbitrary. Given that our goal is to design a consensus protocol that has provable guarantees no matter what the order of message deliveries, it's useful to think of each iteration's message as being chosen by an "adversary" whose sole goal is to foil the consensus protocol.

Ensuring participation through dummy messages. If the message pool M starts out empty, no messages ever get delivered and hence no messages ever get sent. So to get the ball rolling, let's initialize the message pool M with one dummy message $(r, \perp)$ for each node r—because all these messages must eventually be delivered, all nodes will eventually get the chance to participate. You might respond that nodes should be able to participate many times, not just once. This is easily ensured with a convention for what protocols do—let's assume that, whenever a node r receives a message (r, m), it either terminates or sends a dummy message $(r, \perp)$ to the pool M (along with possibly many other messages). This way, every node can participate in the protocol for as long as it wants.

A conspiracy of adversaries. In Lectures 2 and 3 we learned first-hand the challenges of consensus protocol design in the presence of multiple Byzantine nodes—because Byzantine nodes can do literally anything (other than break cryptography), they might well deviate from the intended protocol in a coordinated way, conspiring to foil its design goals.

In the asynchronous model, a conspiracy among Byzantine nodes picks up a new, powerful ally—*adversarial message delivery.* For a protocol to be robust to both Byzantine nodes and adversarial message delivery, it must in particular survive conspiracies between them, perhaps with message delays enabling the shenanigans of the Byzantine nodes. In effect, the actions of all the Byzantine nodes *and* the choices of which messages to deliver are controlled by a single malicious actor. The power of this actor is what makes protocol design in the asynchronous model so challenging.

Interpreting the asynchronous model. Your response to the asynchronous model might be that it doesn't seem very realistic—who is this all-powerful actor who can dictate which messages are received when in the Internet? And that's true. But the point of the asynchronous model is absolutely *not* to faithfully model communication over the Internet. Rather, the point is to avoid making any *assumptions* about how that communication works.

Remember in Lecture 2, when we talked about how the interpretation of Byzantine nodes has evolved with technology over the decades? Back in the 1980s and 1990s (e.g., with IBM replicating a database for higher uptime), researchers in distributed computing thought hard about protocols resilient to Byzantine nodes. They did this not because they were literally worried about malicious actors, but rather because they wanted to avoid any (necessarily limited and controversial) assumptions about how a node with buggy software might behave. Fast forward to the present and the context of blockchains securing billions of dollars, and we very much *do* want to literally model malicious actors whose sole goal is screw up our protocol.

The asynchronous model makes no assumptions about message delivery for the same reasons researchers in the 1980s often made no assumptions about the behavior of faulty nodes—the alternative of imposing specific and hard-to-justify constraints would be much worse.[5]

It's dangerous to put much stock in a protocol whose guarantees are predicated on an overly specific model of "communication over the Internet." Even if that model is valid now, why should it be valid tomorrow? The dream, then, is to design a consensus protocol that works in the asynchronous model, in the absence of any assumptions (other than eventual message delivery). Such a protocol would automatically be interesting, because it would give you the functionality you want even with a barely functioning communication network. Alas, as we'll see, this dream cannot be realized without pulling back some from the extreme lack of assumptions of the asynchronous model.

4 Byzantine Agreement

The FLP impossibility theorem concerns a (famous) consensus problem that we've not yet discussed, *Byzantine agreement*. Before defining it, let's be clear on what we mean by a "protocol."

Protocols. As usual, a protocol is code deployed locally at each node that specifies what the node should do—that is, what messages it should send—as a function of what the node knows. Keep in mind that the asynchronous model is purely event-driven, and so the protocol is invoked at a node upon receipt of a new message. At that moment in time, the

[5]Similarly, there is a long tradition of assessing the performance of an algorithm (MergeSort, Dijkstra's shortest-path algorithm, etc.) through its worst-case running time (over inputs of a given size). This is done not because anyone believes that the input to the algorithm is literally being chosen by a malicious opponent, but rather to encourage the design of "general-purpose" algorithms that do not rely on any overly limiting assumptions about which inputs are the ones that matter.

node knows exactly two things: (i) whatever private input it started the protocol with; (ii) whatever sequence of messages it has received thus far. A protocol therefore specifies, for every possible value of (i) and (ii), the messages that a node should send in response to the most recently received message. We stress that the prescription made by a protocol cannot depend on information that a node does not know (e.g., the message sequences that have been received by other nodes). If two different runs of a protocol result in the exact same sequence of messages getting delivered to a node (and the private input is the same in both runs), then that node will behave identically in the two runs (and in particular will output the same thing either way).

The Byzantine agreement problem. The Byzantine agreement (BA) problem is a single-shot consensus problem, similar to the Byzantine broadcast (BB) problem studied in Lectures 2 and 3. Unlike the BB problem, in the BA problem there is no distinguished sender node—all the nodes play the same role. In the BB problem, the sender is the only node with a private input; in the BA problem, *every* node i has a private input v_i, drawn from some known set V of possible values. (In a blockchain context, v_i might be an ordering of the as-yet-unexecuted transactions that i knows about and V the set of all possible such sequences. For this lecture's impossibility result, we'll only need to consider the case with $V = \{0, 1\}$.) As usual, "private" means that, when the protocol commences, nobody other than node i knows anything about what v_i is (other than that it is some element of V).

Desired protocol properties. What constitutes a "solution" to the Byzantine agreement problem? Like Byzantine broadcast, we're interested in protocols that satisfy three properties: termination, agreement (the safety property), and validity (the liveness property). Termination and agreement are defined exactly as in the Byzantine broadcast problem. The validity property needs to modified to reflect the fact that all nodes have a private input rather than just one. Intuitively, if all the honest nodes begin the protocol with no disagreement among their private inputs, then their outputs should match their inputs (i.e., Byzantine nodes should not be able to trick them into deviating from their common input). Formally:

Desired Properties of a Byzantine Agreement Protocol

1. **Termination.** Every honest node i eventually halts with some output $w_i \in V$.
2. **Agreement.** All honest nodes halt with the same output (no matter what the private inputs are).
3. **Validity.** If $v_i = v^*$ for every honest node i, then $w_i = v^*$ for every honest node i.

As usual, what's hard is getting all the properties simultaneously. Termination and agreement by themselves are trivially achievable (with all honest nodes always outputting a default

value $\perp$), and similarly for termination and validity (with honest nodes outputting their private inputs).

Why a third consensus problem? You might be annoyed that we've introduced yet another consensus problem. Why not prove an impossibility result directly for the BB or SMR problems? (The BA problem is actually the most canonical of the three, but that's not the main reason.)

In the asynchronous model, Byzantine broadcast turns out to be trivially unsolvable.[6] Meanwhile, the BA problem captures the essence of why consensus is impossible in the asynchronous model. In particular, the FLP impossibility result for the BA problem implies the impossibility of SMR in the asynchronous model, which is arguably the result that we really care about.[7]

5 The FLP Impossibility Theorem

Now that we understand the asynchronous model, what we mean by a protocol in this model, and the definition of the Byzantine agreement problem, we're in a position to state the famous FLP impossibility theorem (here "FLP" stands for the three researchers who proved it: Fischer, Lynch, and Paterson). The theorem states that, even with only a single Byzantine node, there is no deterministic protocol that solves the Byzantine agreement problem in the asynchronous model.

Theorem 5.1 (FLP Impossibility Theorem [2]) *For every $n \geq 2$, even with $f = 1$, no deterministic protocol for the Byzantine agreement problem satisfies termination, agreement, and validity in the asynchronous model.*

Several comments:

- Here "deterministic" means that nodes do not locally flip any random coins—the messages sent out by a node in response to a newly received message are completely determined by its private input and the messages it has received thus far.

- The FLP impossibility theorem also has implications for randomized protocols (in which the messages sent out by a node can be a probabilistic function of their private input and received messages). What its proof really shows is that, for every protocol (deterministic or randomized) that is guaranteed to satisfy agreement and validity upon termination, there exists a non-terminating run of the protocol (while still satisfying the eventual delivery property). Thus, no randomized protocol can guarantee all three

[6] This is a good exercise for you to work out. Intuitively, honest non-senders cannot distinguish between a Byzantine sender who is giving them the silent treatment and an honest sender whose messages have been delayed a very long time.

[7] This is another good exercise. Think about how you could take an SMR protocol that guarantees consistency and liveness in the asynchronous model with $f = 1$ and build from it a BA protocol that satisfies termination, agreement, and validity in the asynchronous model with $f = 1$. (Feel free to assume $n \geq 3$.)

of the properties that we want all of the time.[8] The best we can hope for from a randomized protocol that guarantees agreement and validity upon termination are probabilistic guarantees on termination (e.g., a small expected running time, a bounded running time with high probability, and/or a finite running time with probability 1).[9]

- Guaranteeing consensus gets harder as f, the maximum number of Byzantine nodes, grows larger. Thus the most impressive possibility results are the ones with large values of f (the Dolev-Strong protocol being an extreme example). For the same reason, the most impressive impossibility results are the ones that hold even with small values of f. The fact that the FLP impossibility theorem holds even with $f = 1$ is in this sense the strongest result possible![10]

- The statement of Theorem 5.1 above actually undersells the FLP impossibility theorem, which holds even with a single *crash* fault. (As discussed in Lecture 2, a crash fault is the special case of a Byzantine node that follows the protocol honestly up to some moment in time at which someone pulls the node's plug. After that point, the node neither receives nor sends any further messages for the rest of the protocol's run. Such a node cannot, for example, deliberately send out conflicting messages to different nodes.) We'll prove the theorem assuming one Byzantine node; a good exercise for you is to extend the proof (via a couple of minor tweaks) to hold more generally with one crash fault.

- The impossibility result continues to hold under the PKI assumption. (If you think about it, this is an automatic consequence of the previous point.)

- Theorem 5.1 formally separates what is possible in the synchronous and asynchronous models and shows that, as intuition might suggest, consensus is fundamentally more difficult in the latter model.[11]

---

[8]Such a protocol would be analogous to a "Las Vegas" randomized algorithm (e.g., QuickSort), which is one that always terminates with a correct answer but has an unpredictable running time.

[9]There are in fact randomized BA protocols that offer such probabilistic guarantees in the asynchronous model. Practical blockchain protocols tend to escape the FLP impossibility theorem using timing assumptions (as in the partially synchronous model studied in Lecture 6 and the Tendermint protocol covered in Lecture 7) rather than randomization, however. To read more about the history of randomized consensus protocols and practical randomized blockchain protocols (used e.g. by the Helium network) for the asynchronous model, see the HoneyBadgerBFT paper [3] and the references therein. (See also [1] for the subsequent BEAT family of protocols.)

[10]If $f = 0$, the BA problem is solvable even in the asynchronous model. Do you see how to do it?

[11]Under the PKI assumption, the Dolev-Strong protocol shows that the BB problem can be solved in the synchronous model no matter what f is. A good exercise is to show how to use the Dolev-Strong protocol (or any other solution to BB) as a subroutine to solve the BA problem in the synchronous model for all $f < n/2$.

Without the PKI assumption, it's still possible to solve the BA problem in the synchronous model when $f < n/3$ [4]. Because the FLP impossibility result holds for $f = 1$ and all $n \geq 2$, it shows that (with or without the PKI assumption) consensus protocols are fundamentally more powerful in the synchronous model than in the asynchronous model.

- Don't forget: the point of an impossibility result is not to discourage anybody from trying to come up with practical solutions to a problem; it's to property educate everybody about the compromises required (as exemplified by the partially synchronous model in Lecture 6).

So: why is the FLP impossibility theorem true?

6 Proof of Theorem 5.1: Configurations

Like the PSL-FLM impossibility result in Lecture 3 (for Byzantine broadcast in the synchronous model with $f \geq n/3$ and without the PKI assumption), we'll proceed by contradiction. That is, we'll assume that there is in fact a deterministic protocol π for the Byzantine agreement problem that is guaranteed to satisfy termination, agreement, and validity in the asynchronous model with $f = 1$. Deriving a contradiction from this assumption would show that π can't exist. The specific plan is to show that, by virtue of π satisfying agreement and validity upon termination, there will inevitably be cases in which it runs forever (contradicting termination).

Protocol runs as walks in a directed graph. Given a protocol π, we need to exhibit an infinite sequence of things that can happen without the protocol terminating. To make sense of this, let's define a *configuration* as a snapshot of a protocol's trajectory—information sufficient to restart the protocol from where it left off. Precisely, a configuration includes:

- the current state of the message pool M;
- the private input of each of the nodes;
- the sequence of messages received thus far by each of the nodes.

You can think of a protocol's run as a walk through a big (possibly infinite) directed graph, with vertices corresponding to configurations C and edges corresponding to message deliveries $C \xrightarrow{(r,m)} C'$ (Figure 1). Note that whenever a message $(r, m) \in M$ is delivered, the configuration changes in three ways: (i) (r, m) is removed from the pool M; (ii) the delivered message is appended to the end of r's sequence of messages received thus far; and (iii) newly sent messages (by r, as prescribed by π or by r's Byzantine strategy) are inserted into M. Exhibiting a sequence of message deliveries for which π runs forever (contradicting termination) is tantamount to exhibiting an infinite path in this directed graph.

Three types of configurations. The next definition classifies configurations into three categories. Assume that the private input of every node is either 0 or 1 (i.e., $V = \{0, 1\}$).[12]

[12] Remember, we're proving an impossibility result; the fact that we can get away with using only two distinct private inputs makes it that much more impressive. (For a possibility result, you would not want to impose such a restriction on V.)

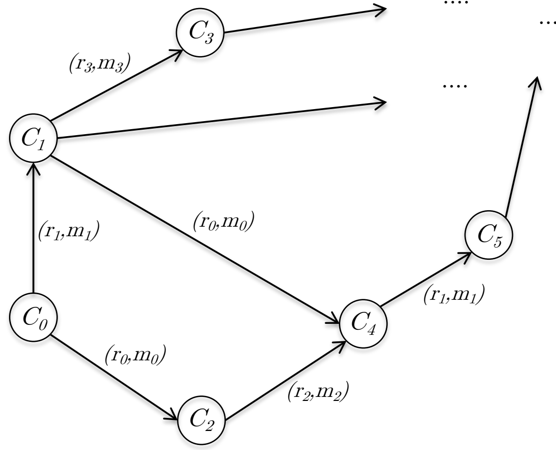

Figure 1: A run of a protocol can be visualized as a walk through a directed graph, with vertices corresponding to configurations and edges to message deliveries.

Because π satisfies agreement (by assumption), whenever π terminates, there are only two possible outcomes: either all the honest nodes output 0 (the "all-zero outcome"), or they all output 1 (the "all-one outcome").

Definition 6.1 (0-, 1-, and Ambiguous Configurations) A configuration is:

- a *0-configuration* if all possible strategies of the Byzantine nodes and all possible sequences of message deliveries lead to the all-zero outcome;
- a *1-configuration* if all possible strategies of the Byzantine nodes and all possible sequences of message deliveries lead to the all-one outcome;
- an *ambiguous configuration* if it is neither a 0- nor a 1-configuration.

Because π satisfies termination and agreement, we can equivalently define an ambiguous configuration as one from which there exist strategies for the Byzantine nodes and a sequence of message deliveries that lead to the all-zero outcome, and also such strategies and such a sequence that lead to the all-one outcome. From a 0- or 1-configuration, the eventual outcome is a foregone conclusion (no matter what the conspiracy of adversaries does); from an ambiguous configuration, the adversary can force whichever outcome it prefers. Definition 6.1 is with respect to a fixed protocol π; for example, a given configuration may be ambiguous with respect to one protocol but not ambiguous with respect to another.

High-level proof plan. The proof will hunt for an infinite path in the directed graph in Figure 1, and specifically for an infinite sequence of ambiguous configurations (with respect to the assumed correct protocol π). This will contradict the assumption that π satisfies termination and complete the proof.

To exhibit such a sequence, we'll use two lemmas, the first playing the role of a base case and the second of an inductive step in a proof by induction.[13] Lemma 7.1 will get us started—it states that there exists a choice of nodes' private inputs so that the corresponding initial configuration C_0 is ambiguous. Lemma 8.1 will then show how to exhibit a new ambiguous configuration C_{i+1} from an old one C_i. Applying Lemma 7.1 once and then Lemma 8.1 over and over again produces an infinite sequence $C_0 \to C_1 \to C_2 \to C_3 \to \cdots$ of ambiguous configurations, which is exactly what we wanted. Neither of these lemmas is obvious. Lemma 7.1 is a bit easier to prove (and it doesn't rely on the full power of the adversary in the asynchronous model), so let's start with that one.

7 Proof of Theorem 5.1: Initial Ambiguity

Here's the formal statement of the lemma that will kick off an infinite sequence of ambiguous configurations.

Lemma 7.1 (An Initial Ambiguous Configuration) *For every deterministic protocol π that satisfies agreement and validity upon termination (with $f = 1$ and some $n \geq 2$), there exists a choice of nodes' private inputs such that the corresponding initial configuration is ambiguous.*

Configurations in general can get pretty crazy (e.g., with billions of messages in the message pool), but there are only 2^n possible initial configurations: at the start of the protocol, the message pool $M = \{(r, \perp)\}_{r=1}^{n}$ is uniquely determined and all nodes have received an empty sequence of messages, so the only degree of freedom is the choice of nodes' private inputs (and with n nodes and all private inputs 0 or 1, there are 2^n such choices). In fact, we'll only need to consider $n-1$ of the 2^n initial configurations to identify an ambiguous one.

Proof of Lemma 7.1: Visualize nodes' private inputs as an n-bit string, with the ith bit indicating node i's private input. Imagine starting from the all-0s string and flipping 0s to 1s one-by-one from left-to-right, ending in the all-1s string. This process generates a total of $n+1$ choices for the private inputs, with corresponding initial configurations $X_0, X_1, X_2, \ldots, X_n$ (Figure 2). (In X_i, the first i nodes have a private input of 1 and the last $n-i$ nodes have a private input of 0.)

Next let's invoke the assumption that π satisfies validity.[14] Validity implies that when all nodes' inputs are 0 (and in particular those of the honest nodes are 0), as they are in the configuration X_0, the protocol must terminate in the all-zero outcome (no matter the strategy of Byzantine nodes and the sequence of message deliveries). In the terminology

[13] Recall that a *lemma* is a technical statement that assists with the proof of a theorem (much as a subroutine assists with the implementation of a larger program).

[14] This is the only part of the proof of Theorem 5.1 that uses this assumption. We use agreement throughout the proof (e.g., in Definition 6.1) to restrict our attention to the all-zero and all-one outcomes. (Remember that either agreement or validity by itself is trivial to obtain upon termination, even in the asynchronous model. So the proof of Theorem 5.1 better use both assumptions about π!)

$$\underbrace{1111.........11}_{i}\underbrace{0000......00}_{n\text{-}i}$$

Figure 2: In the initial configuration X_i, nodes $1, 2, \ldots, i$ have a private input of 1 and nodes $i+1, i+2, \ldots, n$ have a private input of 0.

of Definition 6.1, the initial configuration X_0 must be a 0-configuration. Invoking validity again, by the same argument, the initial configuration X_n (in which all nodes' inputs are 1) must be a 1-configuration.

We claim that one of the intermediate configurations $X_1, X_2, \ldots, X_{n-1}$ must be an ambiguous configuration (in which case, we'll be done with the proof). To see this, consider the smallest value of $i \geq 1$ such that X_i is *not* a 0-configuration. (This value must exist because, if nothing else, the choice $i = n$ would meet the criterion.) By the choice of i, X_{i-1} must be a 0-configuration (otherwise, we could have taken i to be smaller).

If X_i is an ambiguous configuration, we're done. The only worry is that X_i might be a 1-configuration—i.e., that flipping the private input of node i from 0 to 1 jumps directly from a 0-configuration to a 1-configuration. Intuitively, such a "pivotal" private input should sound implausible—if node i happens to be Byzantine, it is fully capable of hiding its private input from the rest of the nodes and keeping them in the dark. To make that idea precise, suppose node i is Byzantine and consider two possible strategies for it:

(i) any strategy that forces the all-one outcome (which must exist, given that X_i is not a 0-configuration);

(ii) the strategy in which it pretends its private input is actually 0 and otherwise follows the protocol π honestly.

With strategy (ii), the trajectory of the protocol π is exactly the same as its trajectory from the initial configuration X_{i-1} when all nodes follow π honestly. (All honest nodes see the exact same sequences of messages either way, and hence operate identically in both cases.) Because X_{i-1} is a 0-configuration (by our choice of i), the protocol must terminate in the all-zero outcome. We conclude that the Byzantine node i has the option of forcing either the all-zero or the all-one outcome from X_i, and thus X_i is indeed the ambiguous configuration that we're looking for.[15] ■

8 Proof of Theorem 5.1: Reduction to Lemmas 7.1 and 8.1

Let's move on to the notorious second lemma, which is fairly similar to the first one but definitely a bit trickier. This lemma acts like an inductive step in a proof by induction,

[15] How would you tweak the proof so that Lemma 7.1 holds even with just one crash fault (as opposed to one Byzantine fault)?

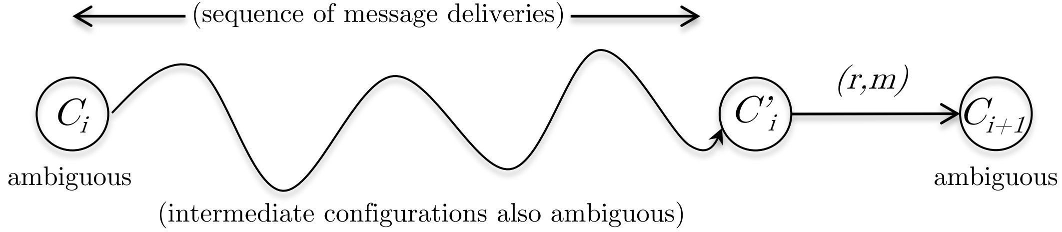

Figure 3: Lemma 8.1: extending a sequence of ambiguous configurations (from C_i to C_{i+1}) while delivering the message (r, m) last.

in that it takes as input a sequence of ambiguous configuration and outputs a longer such sequence. The two lemmas can then be used to exhibit the desired infinite sequence of ambiguous configurations: Lemma 7.1 gets the sequence started, and applying Lemma 8.1 over and over again produces the rest of it.

Statement of the second lemma. Actually, the argument above misses a subtle point. That subtle point is the reason the statement of the second lemma is perhaps more complex than you were expecting (Figure 3):

Lemma 8.1 (Extending an Ambiguous Sequence) *Fix a deterministic protocol π that satisfies agreement and validity upon termination (with $f = 1$ and some $n \geq 2$). Let C_i denote an ambiguous configuration and (r, m) a message in C_i's message pool. Then, there exists a sequence of message deliveries such that:*

(i) the last step of the sequence is the delivery of (r, m);

(ii) the end of the sequence is an ambiguous configuration C_{i+1}.[16]

You might have been expecting a simpler statement: for every ambiguous configuration C_i, there exists a message to deliver resulting in another ambiguous configuration C_{i+1}. What's up with forcing the lemma to work with arbitrarily chosen message (r, m) that's hanging out in C_i's message pool?

The simpler statement is trivially true: start from the initial ambiguous configuration C_0 promised by Lemma 7.1 and then deliver an infinite sequence of dummy messages (messages of the form $(r, \perp)$, see Section 3). The dummy messages don't do anything, so all of the configurations produced will be ambiguous. Unfortunately, iterated application of this simpler statement yields a sequence in which the adversary never delivers any (non-trivial) messages. Of course consensus is impossible if we allow an adversary to do this! This issue is exactly why, in the definition of the asynchronous model in Section 3, we insisted on the

[16]Sanity check: given that C_i and C_{i+1} are both ambiguous, what can we say about the configurations between them? If you think about it, ambiguity can be resolved (with a message delivery forcing the final outcome and triggering a transition from an ambiguous configuration to a 0- or 1-configuration) but can never be introduced (e.g., once in a 0-configuration, always in a 0-configuration). Thus, by virtue of C_{i+1} being ambiguous, we can conclude that all of its predecessors in the sequence must likewise be ambiguous.

constraint that every message sent to the message pool must eventually be delivered to its intended recipient.

The extra complexity in the statement of Lemma 8.1 addresses this exact issue, allowing us to produce an infinite sequence of configurations that satisfies the eventual delivery constraint. Formally, here's the proof that Lemma 7.1 and (the as-yet-unproven) Lemma 8.1 in tandem imply the FLP impossibility theorem:

Proof of Theorem 5.1: Fix a deterministic protocol π that satisfies agreement and validity upon termination (with $f = 1$ and some $n \geq 2$). No prizes for guessing the first step: invoke Lemma 7.1 to choose an initial configuration C_0 that is ambiguous (which must exist).

Presumably we then want to invoke Lemma 8.1 over and over. Each invocation asks us to pick a message (r, m) in the current message pool—how should we choose? Putting Lemma 8.1 aside for a moment, imagine we didn't care about retaining ambiguity and just wanted to make sure that every message eventually gets delivered. An easy solution would be first-in first-out (FIFO): always deliver the message in the message pool that was sent the largest number of iterations ago (even if it forces a transition from an ambiguous configuration to a 0- or 1-configuration). Then, whenever a message is added to the existing message pool M, we know that it will be delivered $|M|$ iterations later (here $|M|$ denotes the number of messages in the pool). Because M is always finite (we only allow a node to add a finite number of messages in a single iteration), every message gets delivered after a finite number of iterations.

Lemma 8.1 is phrased so that we can split the difference between the trivial solution that guarantees never-ending ambiguity but not eventual delivery (i.e., deliver only dummy messages) and the FIFO solution that guarantees the latter but not the former. The key idea is to simulate FIFO message delivery as closely as possible subject to retaining ambiguity. Precisely, here's how we'll define our sequence of ambiguous configurations:

- define C_0 as the ambiguous configuration promised by Lemma 7.1;
- for $i = 0, 1, 2, \ldots$:
  - let (r_i, m_i) denote the oldest message in C_i's message pool;
  - define C_{i+1} as the ambiguous configuration promised by Lemma 8.1 (with respect to C_i and (r_i, m_i)).

This procedure constructs a sequence of sequences—there could be a billion intermediate configurations between some C_i and C_{i+1} (Figure 4)—but whatever, the concatenation of these sequences is itself a sequence of configurations. The sequence between C_i and C_{i+1} is effectively stalling (delivering whatever messages it wants, preserving ambiguity) up to the point at which it can deliver the message (r_i, m_i) without resolving the ambiguity.

By Lemma 7.1 and 8.1, all of the C_i's—and hence, also all of the intermediate configurations (see footnote 16)—are ambiguous configurations. The eventual delivery constraint is satisfied by this sequence for the same reason that FIFO delivery satisfies it: if a message (r, m) gets added to a message pool M at or before a configuration C_i, it is guaranteed

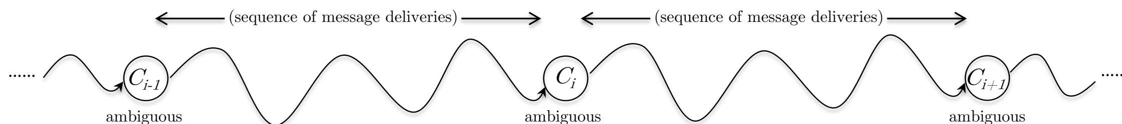

Figure 4: Repeated applications of Lemma 8.1 produces an arbitrarily long sequence (equivalently, sequence of sequences) of ambiguous configurations.

to be delivered by configuration $C_{i+|M|}$ at the latest. (Each invocation of Lemma 8.1 will deliver at least one of the $|M|$ messages that preceded (r, m). After at most $|M|$ invocations, (r, m) will be first in line.)[17] ■

Now we know that Lemma 7.1 is true and that Lemmas 7.1 and 8.1 imply the FLP impossibility result. The final order of business is to prove Lemma 8.1.

9 Proof of Theorem 5.1: Extending an Ambiguous Sequence

To prove Lemma 8.1, fix a deterministic protocol π that satisfies agreement and validity upon termination (with $f = 1$ and some $n \geq 2$), an configuration C_i, and a message (r, m) belong to C_i's message pool. We need to show that it's possible to eventually deliver (r, m) while retaining ambiguity (possibly delivering a billion other messages in the meantime).

9.1 Proof Setup

Next we classify configurations into three categories. Like the classification in Section 6, this one will be defined with respect to the fixed protocol π. Unlike the one in Section 6, it will also be defined with respect to the fixed configuration C_i and message (r, m).

If delivering the message (r, m) at the configuration C_i happens to lead directly to another ambiguous configuration, then we're done and there's nothing more to prove. So suppose delivering (r, m) immediately would lead to a 0-configuration, with the eventual outcome forced to be the all-zero outcome. (The argument for the case in which it leads to a 1-configuration is exactly the same, reversing the roles of 0 and 1.)

Here are the additional three categories of configurations that we're going to need:

Definition 9.1 (0^*-, 1^*-, and Ambiguous* Configurations) Let C be a configuration reachable from C_i via the delivery of a sequence of messages different than (r, m). The configuration C is:

- a 0^*-*configuration* if delivering (r, m) at C leads to a 0-configuration;

[17] The message (r, m) might well be delivered much earlier, if some invocation of Lemma 8.1 tasked with delivering an earlier message (r', m') needs to deliver (r, m) as an intermediate message en route to safely delivering (r', m').

- a 1^*-*configuration* if delivering (r, m) at C leads to a 1-configuration;

- an *ambiguous** *configuration* if delivering (r, m) at C leads to an ambiguous configuration.

Because every configuration is either a 0-, 1-, or ambiguous configuration, every configuration that is reachable from C_i without the delivery of message (r, m) must be either a 0^*-, 1^*-, or ambiguous* configuration. A 0^*-configuration could be either a 0-configuration (with the delivery of (r, m) then immaterial to the eventual outcome) or an ambiguous configuration (with the delivery of (r, m) cutting the adversary off from all its avenues to the all-one outcome). An ambiguous* configuration must be ambiguous—the special case of an ambiguous configuration that remains ambiguous after the delivery of (r, m).

With this new terminology, we can crisply rephrase our goal and one of our standing assumptions:

- conclusion of Lemma 8.1: there exists an ambiguous* configuration (i.e., it's possible to deliver messages so that the subsequent delivery of (r, m) leads to an ambiguous configuration);

- assumption: C_i is a 0^*-configuration. (If C_i is an ambiguous* configuration, there's nothing to prove. If it's a 1^*-configuration, exchange the roles of 0 and 1 in the rest of the proof.)

9.2 Hunting for a Non-0^*-Configuration

Hunting via breadth-first search. We mentioned in Section 1 that the proof plan for Theorem 5.1—exhibiting an infinite sequence of ambiguous configurations—can be visualized as hunting for an infinite path in a big directed graph, with each vertex corresponding to a configuration and each edge corresponding to a transition caused by the delivery of a single message (Figure 1). Now imagine doing breadth-first search from the vertex corresponding to the initial ambiguous (and 0^*-)configuration C_i, with the twist that the search ignores any edges that correspond to the delivery of the message (r, m). By Definition 9.1, every configuration encountered in this search is either a 0^*-, 1^*-, or ambiguous* configuration.

Escaping the land of 0^*-configurations. This search must at some point encounter a configuration that is *not* a 0^*-configuration. For if it never left the land of 0^*-configurations, then whenever the message (r, m) might be delivered, it would necessarily lead to a 0-configuration. The adversary must deliver the message (r, m) eventually (by the constraints of the asynchronous model), and whenever it does, it would force the all-zero outcome. But that means the all-zero outcome is already forced at the configuration C_i, contradicting our assumption that C_i is ambiguous.

A candidate ambiguous* configuration. We still have to rule out the possibility that this breadth-first search only ever encounters 0^*- and 1^*-configurations. To do this, consider

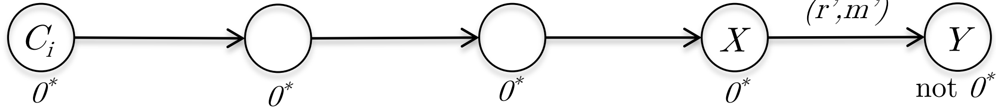

Figure 5: Let Y denote the non-0^*-configuration that can be reached from C_i via the delivery of the fewest number of messages (none of which are (r, m)), and X its predecessor configuration on this shortest path.

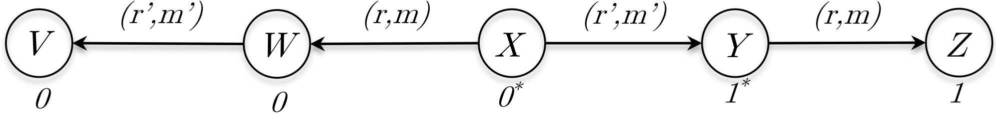

Figure 6: If Y is a 1^*-configuration, the order of the delivery of the messages (r, m) and (r', m') starting from configuration X dictates whether the protocol halts with the all-zero or the all-one outcome.

the first time the search discovers a non-0^*-configuration Y (and it must, eventually). Equivalently, define Y as the nearest non-0^*-configuration to C_i (meaning the fewest number of message deliveries needed to reach it, breaking ties arbitrarily). Let X denote Y's predecessor configuration on the $C_i \rightsquigarrow Y$ path by which the search discovered Y, and (r', m') the last message delivered along this path (triggering the $X \mapsto Y$ transition); see Figure 5. The message (r', m') must be different from (r, m), as the search only considers paths that do not deliver (r, m); for the same reason, and because (r, m) belongs to C_i's message pool, it must also belong to both X's and Y's message pools. The configuration X might or might not be the same as C_i (depending on whether there is some message in C_i's message pool whose immediate delivery would lead to a non-0^*-configuration). In any case, because Y is the closest non-0^*-configuration to C_i and X is Y's predecessor, X must be a 0^*-configuration.

We can complete the proof by arguing that Y cannot be a 1^*-configuration (and thus must be the ambiguous* configuration that we seek). Toward a contradiction, suppose that Y is in fact a 1^*-configuration and zoom in on the configurations X and Y, with the delivery of (r', m') at X leading to Y (Figure 6). As 0^*- and 1^*-configurations, respectively, the delivery of (r, m) at X or Y would lead to a 0-configuration (call it W) or a 1-configuration (call it Z), respectively. For kicks, we can also think about delivering the still-undelivered message (r', m') at W (as opposed to at X, as we did originally), to reach still another configuration V. Because W is a 0-configuration (with the all-zero outcome forced no matter what), so is V.

A pivotal pair of messages. The key takeaway from Figure 6 is that, starting from the configuration X (whose message pool contains both (r, m) and (r', m')):

(P1) delivering message (r, m) followed by (r', m') results in a 0-configuration (namely, V);

(P2) delivering message (r', m') followed by (r, m) results in a 1-configuration (namely, Z);

In other words, starting from the configuration X, the relative order in which the messages (r, m) and (r', m') are received by their recipients is pivotal information and completely dictates whether the protocol's final outcome is the all-zero or the all-one outcome (flip their order and you'll flip the outcome). Perhaps you can sense the looming contradiction? We'll prove it using two cases—the first one direct and easy (and with no Byzantine nodes needed), the second one similar to our argument in the proof of Lemma 7.1.

Case 1: $r \neq r'$. If the two messages in question are bound to different recipients, then no node is aware of the relative order in which they were received. (Remember: all a node knows is its private input and the sequence of messages it has received itself, and its behavior is completely determined by this knowledge. It does not directly know anything about the sequences of messages received by other nodes.) Flipping the order in which the messages are received does not affect the sequence of messages received at any node (or any private inputs), so all nodes operate identically either way (and in particular, output the same thing). This contradicts statements (P1) and (P2).

Case 2: $r = r'$. If the two messages are bound to the same recipient r, then node r and node r alone knows the relative order in which the messages were delivered. (The analog in the proof of Lemma 7.1 is the alleged node with the pivotal private input.) But if r happens to be the Byzantine node, it is fully capable of hiding from the other nodes the true order in which it received (r, m) and (r', m'), keeping them in the dark. To make that idea precise, suppose node r is Byzantine and consider two possible strategies for it:

(i) follow the protocol π honestly;

(ii) pretend that it received the messages (r, m) and otherwise follow π honestly.

Nodes other than r cannot distinguish between and therefore operate identically in two scenarios: node r received message (r, m) before (r', m') and is acting honestly (strategy (i)); node r received (r', m') before (r, m) and is using strategy (ii). This contradicts the facts that the all-zero outcome is forced in the first scenario (by (P1)) and the all-one outcome if forced in the second scenario (by (P2)). This contradiction implies the incorrectness of our assumption that the configuration Y was not an ambiguous* configuration; hence Y is exactly the type of configuration we were hunting for.[18] ■

10 Discussion

Congratulations! You've survived the not-at-all easy proof of the famous FLP impossibility theorem, which puts you in the rarified company of distributing computing experts. Before moving on to climb further mountaintops in the next lecture, let's not forget the forest for the trees.

[18]How would you tweak the proof so that Lemma 8.1 holds even with just one crash fault (as opposed to one Byzantine fault)?

First, remember that the point of impossibility results is not to discourage anybody from trying to come up with really cool consensus or blockchain protocols. Rather, the point is to educate everybody about the design choices that must be made, the compromises that must be accepted. For example, the FLP impossibility result (and its extension to the SMR problem, see footnote 7) specifically teaches you that every blockchain consensus protocol must choose between consistency (all nodes stay in sync) and liveness (transactions keep getting processed) when the network is under attack. This choice is a top-level node in the decision tree of blockchain protocol design, with "BFT-type" protocols (like Tendermint in Lecture 7) favoring consistency when under attack and longest-chain protocols (introduced in Lecture 8) favoring liveness. No matter how smart you are, you'll never come up with a protocol that guarantees the best of both worlds.

Second, impossibility results clarify which assumptions matter. For example, Lectures 2 and 3 showed that the PKI assumption (and the existence of cryptography) really matters (at least in the synchronous model), with results achievable under this trusted setup assumption that are provably impossible without it. Similarly, the FLP impossibility result confirms and formalizes the intuition that the synchrony assumption of Lectures 2 and 3 really matters—the degree of network reliability fundamentally changes what consensus protocols can accomplish.

Third, impossibility results guide you toward the minimal assumptions necessary for the existence of protocols with provable guarantees. Next lecture, Lecture 6, discusses the partially synchronous model, which was explicitly conceived as a compromise between the unrealistic synchronous model and the overly demanding asynchronous model. It's not clear anyone would have come up with that "sweet spot" model—arguably the most important one for assessing the basic properties of blockchain consensus protocols—without the guidance provided by the FLP impossibility result.

Finally, speaking as a theoretician, how cool are these proofs? The impossibility results from the past few lectures are among the greatest hits of computer science. And even though our eyes are focused primarily on the future in this lecture series, it's also important to celebrate the past—especially when the past remains so practically relevant. Just as many find it spiritually nourishing to experience fine works of art, perhaps at least a subset of you will have a similar feeling when internalizing these proofs by the brilliant computer scientists who have preceded us.

References

[1] S. Duan, M. K. Reiter, and H. Zhang. BEAT: Asynchronous BFT made practical. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 31–42, 2018. URL: `https://www.csee.umbc.edu/~hbzhang/files/beat.pdf`.

[2] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, 1985. URL: `https://groups.csail.mit.edu/tds/papers/Lynch/jacm85.pdf`.

[3] A. Miller, Y. Xia, K. Croman, E. Shi, and D. Song. The honey badger of BFT protocols. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, pages 31–42, 2016. URL: `https://eprint.iacr.org/2016/199.pdf`.

[4] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980. URL: `https://lamport.azurewebsites.net/pubs/reaching.pdf`.